Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Translating Hand-Drawn Map Sketches to Digitalized Fantasy Maps

Ricardo de Castro Giometti Santos

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Alexandre Ricardo Soares Romariz

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Translating Hand-Drawn Map Sketches to Digitalized Fantasy Maps

Ricardo de Castro Giometti Santos

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Alexandre Ricardo Soares Romariz (Orientador)
FT/UnB

Prof. Dr. Eduardo Peixoto Fernandes da Silva
FT/UnB

Prof.a Dr.a Mylene Christine Queiroz de Farias
FT/UnB

Prof. Dr. João Luiz Azevedo de Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 5 de junho de 2023

Dedication

I dedicate my work to all those who have played a role, big or small, in my academic and personal development

Acknowledgements

I would like to express my deepest appreciation to my family, whose love, care, and understanding have been the pillars that have sustained me during these challenging years of study. Their constant nurturing and support have been instrumental in my accomplishments. Within my family, I want to extend a special thank you to my father. Despite being far away, his tireless sacrifices, dedication, and unending encouragement have been the driving force behind my achievements. His unwavering belief in my abilities has pushed me to overcome every obstacle and always strive for excellence. I am also immensely grateful for my girlfriend, who has been a constant source of emotional support. Her belief in me, coupled with her assistance and willingness to lend a listening ear during my moments of frustration have been invaluable. Finally, I would also like to extend my sincere thanks to my project advisor, whose guidance and advice have played a vital role in shaping and refining my work.

Resumo

Neste trabalho, utilizo Redes Geradoras Adversariais para transformar mapas desenhados à mão em suas versões fantasiosas. Utilizando a biblioteca tkinter, em Python, desenvolvo um aplicativo de desenho no qual um usuário pode rapidamente rabiscar um mapa e submetê-lo a um Modelo Gerador, chamado Pix2Pix. Dessa forma, ele obtém instantaneamente a ajuda da inteligência artificial para adicionar mais detalhes à sua ideia, aumentando assim a velocidade e eficiência. Embora eu tenha me concentrado principalmente na aplicação dessa ideia em mapas desenhados a mão de RPG de mesa, o conceito pode ser estendido a outros domínios, como, por exemplo, videogames e até plantas arquitetônicas. Ao longo deste trabalho, explico a teoria por trás do modelo e apresento os resultados obtidos ao utilizá-lo em três experimentos diferentes. No primeiro experimento, uma versão pré-treinada do modelo é utilizada para avaliar suas capacidades gerais. No segundo, utilizo um conjunto personalizado de dados feito à mão para treinar o modelo, e apresento o aplicativo desenvolvido para melhorar a experiência do usuário. No terceiro e último, utilizo um pequeno conjunto de dados que contém imagens de alta resolução para avaliar a capacidade de aprendizagem do modelo. Concluindo o trabalho, apresento minhas próprias opiniões sobre os resultados e dou uma visão sobre novos modelos que podem ser utilizados para melhorar a qualidade da imagem.

Palavras-chave: GANs, Pix2Pix, Jogos, Criação de Mapas, Redes Geradoras, Tradução de Imagem para Imagem

Abstract

In this work I use Generative Adversarial Networks to transform hand-drawn maps into their computerized fantasy versions. With Python's tkinter GUI library, I develop a drawing application in which a user can quickly sketch a map and submit it to a Generative Model called Pix2Pix. In this manner, he instantly gets the help of artificial intelligence to add more detail to his idea, increasing speed and efficiency. While I mainly focus on the application of this idea to hand-drawn tabletop RPG maps, the concept of translating drawings to images can be extended to other domains such as video games and even architecture blueprints. Throughout this work I explain the theory behind the model and present the results obtained when utilizing it in three different experiments. In the first, a pre-trained version of the model is used to asses its overall capabilities. In the second, I use a custom hand-made dataset to train the model and I showcase the developed application to improve user experience. In the last, I use a small dataset that contains high resolution images to asses the model's learning ability. Concluding the work, I give my own personal opinions over the results and give a intuition of newer models that can be used to improve image quality.

Keywords: GANs, Pix2Pix, Games, Map Creation, Generative Networks, Image-To-Image Translation

Contents

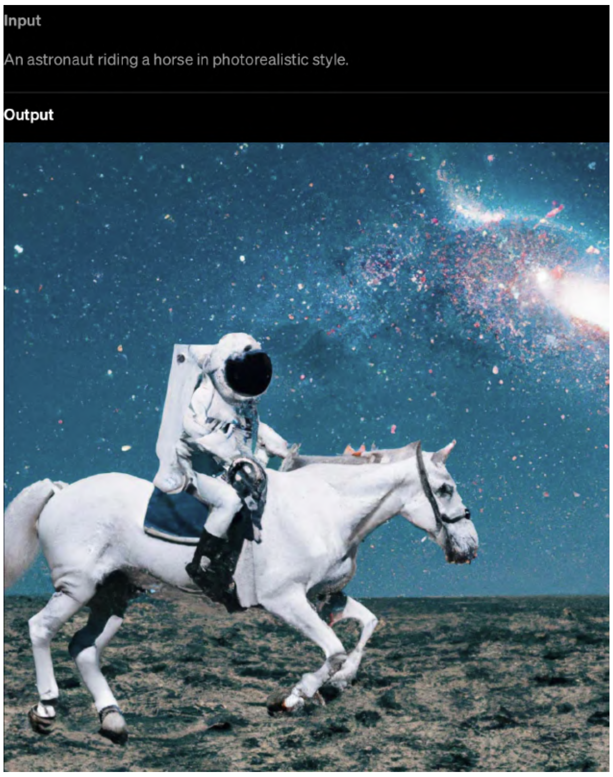
List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the field of artificial intelligence and deep learning has witnessed significant advancements, mainly due to the emergence of a new class of learning models known as Generative models. As their name suggests, these are models that have the ability of generating data. Among various applications, they have revolutionized the task of image creation by enabling the automatic generation of visually appealing and high-quality content. In this space, both DALL-E and Midjourney, have been gathering significant attention for their impressive results.


Input

An astronaut riding a horse in photorealistic style.

Output

Figure 1.1: Dalle-2 example prompt and output, from [1].

Both, using their own specialized methods, have the ability to create artwork that satisfies a specific request given as textual prompt [8], [9]. As these models continue to gain praise for their contributions to the world of AI-driven creativity, another related task is becoming increasingly apparent: Image to Image translation.

Image translation is a field of research within computer vision that studies the possibilities of transforming images from one domain to another while preserving relevant content and structure. Traditionally, image translation tasks relied on image processing algorithms and techniques such as image denoising or edge detection to be solved. However, advancements in AI have opened doors to face more complex problems, driving the development of innovative solutions.

Among the various generative-based architectures designed to tackle this problem, Pix2Pix has gained significant attention and popularity since its introduction in 2016. It has demonstrated remarkable results in image-to-image translation tasks, where it established itself as a go-to method in these types of problems and sparked further research in the field of image translation using generative models.

A notable application of the Pix2Pix architecture lies in its ability to transform drawings and sketches into pictures. Figure 1.2 presents a demonstration where the Pix2Pix model used the sketch as base to generate an image of a cat. This unique application empowers artists by allowing them to actively participate in the creative process. By increasing the participation of the user in the process, this application bridges the gap left by prompt-based models by maintaining a sense of connection and ownership over the final results.

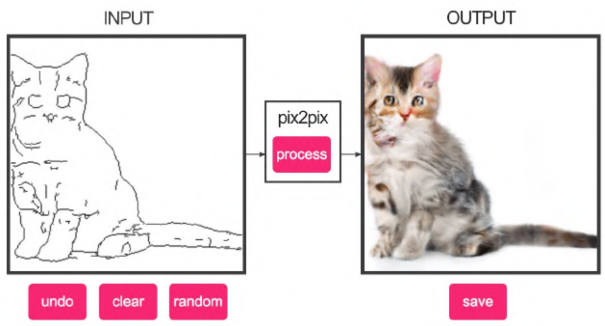


Figure 1.2: Edges to Cat example demo, from [2]

1.1 Motivation

Dungeons and Dragons, commonly known as DnD, is a fantasy pen and paper role-playing game that was first published in 1974 [10]. Role-playing game, or RPG, is a gaming genre where participants assume the personality of made up characters in a fictional setting.

In the game, participants must act within their character narrative and explain what their character would like to do in different situations. These actions can succeed or fail, according to a predefined set of rules and guidelines. Within the rules, players have the freedom to improvise and their actions are only limited by the power of imagination. The publication of DnD is commonly referred to as the beginning of modern role-play gaming.

In DnD, player characters embark on adventures within a fantasy scenario. What makes the game unique is that everything about the players' characters is created by themselves: their goals, abilities and flaws. When playing the game, each participant decides their character's actions based on these definitions. The players' characters join into an adventure group called a party, and together solve mysteries, engage in battles and explore the world they are set in.

Figure 1.3: Dungeons & Dragons gameplay

In order for the game to flow, one of the participants assumes the role of Game Master (GM), also called the Dungeon Master (DM). The GM acts as the game's storyteller and referee, setting the scenes the players are in, presenting the problems and challenges they face and assuming the personality of every inhabitant of the game world, called NPCs, or non-playable characters, that the player characters can interact with and engage in conversations.

The gameplay is divided into sessions of play and has indefinite length. During each session, players face many challenges and completing many adventures designed by the GM. The set of all related game adventures form what is known as a campaign.

The game requires three component to be played.

- The three core rulebooks The Player's Handbook, the Dungeon Master's Guide and the Monster Manual, that outline the core rules of the game.
- Character Sheets for players to take notes and record specific attributes or items their character has.
- A set of dice used to determine the success, or failure, of a character's action.

Beyond that, many players like to use additional accessories such as miniature figures and maps that enhance the player experience and immersion.

Over the years, alongside advancements in electronic components, the RPG genre evolved towards new media format. Powerful graphical cards enabled the genre to expand onto computer screens and televisions, where immersive scenarios and lifelike NPCs could be rendered in real time. These visual enhancements, that each time more accurately captured the fantastical ambiance of the adventure, attracted the interest and admiration of the gaming community.

Despite the expansion to new media, the interest in traditional pen and paper format remained. As the on-screen visuals improved, the interest in reproducing a similar ambiance on pen and paper games emerged and multiple tools to design maps and character images were created to fill this need. Despite these tools offering the ability to create impressive visuals, they often have a steep learning curve and are time-consuming. Just creating a quick map or character photo can take multiple hours scattered throughout many days.

1.2 Goals

In this context, the difficulty of easily creating fantasy maps is a problem faced within the RPG community. Due to the difficulty in using online tools, players are left with the option of creating the game world maps using pen and paper. Although most of the time this satisfies the immediate need for a basic spacial representation, it falls short of providing an immersive experience.

Not only that, but creating maps and assets is also a problem faced in the entire gaming industry. In order to create the previously mentioned ambiance that exists in Video Games, developers stay thousands of hours coding each detail that will be rendered onscreen. Due to this, the release of new games can take months and even years.

In this work, I use Artificial Intelligence to address this ongoing problem. More specifically, I have the objective of transforming traditional pen-and-paper sketches, scanned with a printer, into fantasy maps. Additionally, I aim in developing a drawing environment in which players can sketch their map ideas and have the model interpret these sketches and output a faithful fantasy map representation of what the person envisioned with the sketch.

1.3 Project Structure

The remaining portion of this work is divided into three additional chapters.

Chapter 2 introduces the fundamental concepts required to understand the final model used in the project, called Pix2Pix. It covers an explanation of generative models and their distinction from traditional classifiers. Beyond that, it explains how the GANs framework innovated the field of generative AI and explores enhancements, such as conditional generation, and fixes that aimed in addressing issues such as training instability. Additionally, the chapter discuesses various methods to evaluate generative models and presents algorithms used to create training datasets.

Chapter 3 provides details of the development process. Each section in this chapter represents an experiment that builds upon the successes and failures of the previous. It delivers detailed explanation for the procedures employed in training and testing the model, with a special focus on explaining how the data used for training was created and how it affected the model's output. The chapter also demonstrates the custom drawing tool developed to further enhance the testing experience. Throughout the chapter, conclusions and insights are presented, highlighting key observations made during each iteration of the experiment.

Chapter 4 provides an overview and conclusion of the entire experiment. It assesses if the goals for the project were achieved as well as propose other models and techniques that can be explored in the future to further improve the project's overall performance.

Chapter 2

Theory

In recent years, research in the field of Artificial Intelligence has surged in wave-like patterns. A groundbreaking discovery ignites a wave of excitement among researchers, leading to great expectations and paving the way for a network of new investigations. We are currently experiencing such phenomenon.

Although the use of AI for solving classification problems has become well-established, its application in tasks involving generative behavior had previously shown limited progress. However, there has been a recent upturn in advancements in this area [11]. Novel models are being actively researched and utilized as the foundation for even more refined and sophisticated approaches.

This has transformed the online landscape to withhold a flood of AI-generated content such as images, videos, music and even small to large articles.

2.1 Data Classifiers

Andrew Y. Ng and Michael I. Jordan, define two learning models [12]: Generative and Discriminative. They provide a comparison of these models in terms of their approach to solving classification tasks.

As they explain, Discriminative models have the goal of learning how to map an input x to a known label or class y. These models achieve this by directly learning how to calculate the conditional probability $p(y|x)$, shorthand for what is the probability of y given x.

Generative models, on the other hand, have the goal of learning the direct mapping of an input x to a label y. In other words, it learns a model of the joint probabilities $p(x, y)$

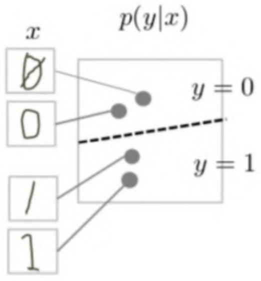


Figure 2.1: Discriminative Classification, from `developers.google.com`

of the input and uses Bayes Theorem:

$$P(y|x) = \frac{P(x,y)}{P(x)} \tag{2.1}$$

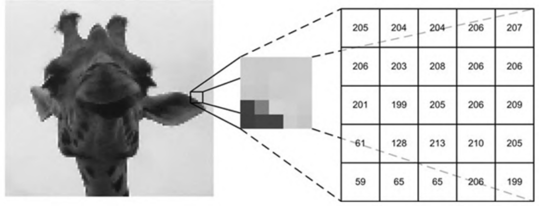
to calculate $p(y|x)$. This means that Generative classifiers solve a more general modeling problem before classification.

In practical terms, Discriminative models will output how likely your input is to each of its known labels while Generative models will output a possible input that can be mapped to a specified label.

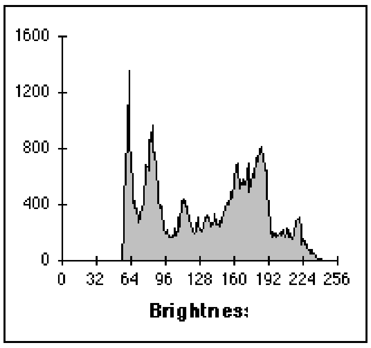
While [12] mainly focuses on explaining why Discriminative models are a better fit for classification tasks, it can be infered that Generative models offer a solution to the problem of creating new data.

2.2 Generative Models

Generative Models operate on the principle that every image can be interpreted as a probability distribution of it's pixels.



(a) Image's Pixels



(b) Histogram of the Pixels

The core concept behind Generators is to establish a mapping of similar images to a shared spatial probability.

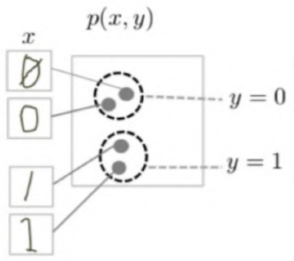


Figure 2.3: Generator Spatial Mapping, from `developers.google.com`

By sampling values from a random distribution, the Generator can then remap these values to another distribution that fits within the learned spatial distribution. In this manner, it effectively generates new data with desired characteristics.

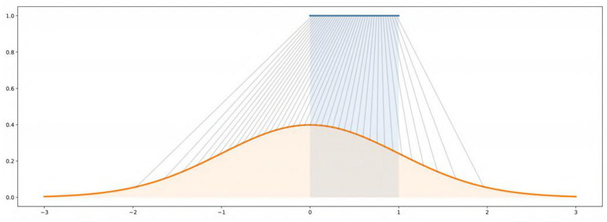


Figure 2.4: Distribution Remapping, from `deeplearningbook.com.br`

2.3 Adversarial Networks

In the context of generative models, the goal is to learn a model that can generate fake data samples that seamlessly resemble a given distribution set. However, achieving this has proven to be quite challenging, primarily due to the difficulty in directly approximating complex probabilistic computations that arise in maximum likelihood estimation strategies.

Another challenge arises from the difficulty in leveraging the piecewise linear units that are commonly used in neural networks. Backpropagation, the algorithm used to update network parameters, relies on the flow of gradient information through the network layers. However, in generative networks, the errors between each linear unit are difficult to minimize, making the optimization process problematic.

In 2014, Ian J. Goodfellow *et al* [3], proposes a novel training strategy that manages to get around these problems.

He introduces an adversarial network framework in which two neural networks are put to compete against each other. The first of those networks is called the Generator and the other is the Discriminator.

The Generator network's job is to create fake data and the Discriminator's job is to determine whether a given data input sample came from a real (1) data distribution or from a fake (0) generated distribution.

The training framework presents itself as a competition game that simultaneously improves both networks: the Generator creates data increasingly more similar to the real, and, the Discriminator distinguishes what is fake or real.

In this sense, this training strategy is commonly compared to a group of counterfeiters printing money (the generator creating data) trying to fool the police that is looking out for fake money (discriminator figuring if data is real of fake). However, in this case, instead of apprehending the counterfeiters, the police provides feedback to the counterfeiters on how they can improve their craft and create better replicas of the real money.

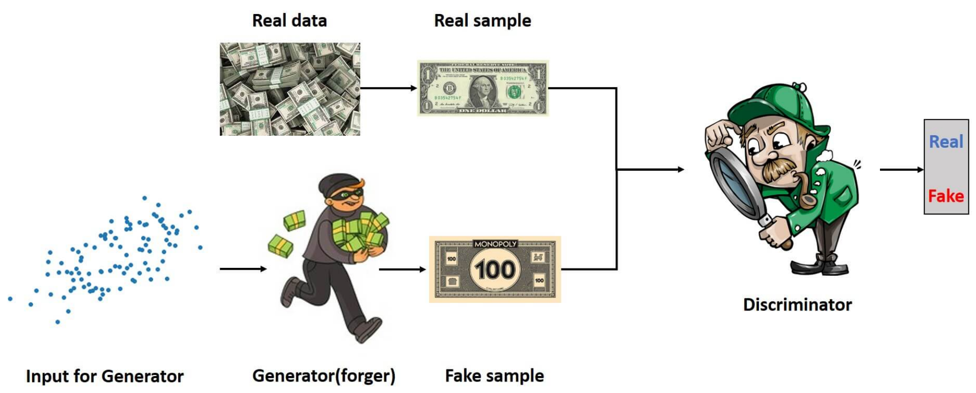


Figure 2.5: GANs compared to a group of counterfeiters and a detective, from `datahacker.rs`

Our desired optimum result happens when the Discriminator cannot distinguish fake from real, meaning, the generator creates data with characteristics identical to the real.

By employing this strategy, Goodfellow sidesteps the need to directly optimize the Generator by implicitly learning the complex probability distribution without directly observing the target data.

2.4 Adversarial Training of the Generator

Neural networks are function approximation algorithms. They operate like a black box, where they estimate an unknown underlying function using a subset of data provided by observations in the domain. During the training process, through a technique called back-propagation, the error between the predicted outputs and the expected outputs is calculated and used by the neural network to update it's parameters in order to minimize this error [13]. Through many iterations of error calculation and parameter optimization, the network progressively adapt its parameters to effectively map those of the underlying data.

The Generator is a neural network represented by the function $G(z; \theta_g)$, where θ_g are learnable parameters. It receives a random noise vector z as input and outputs data $x_g = G(z)$. The vector z is created by sampling data points from a random Gaussian distribution p_z. The x_g output from the generator is treated as a sample from the Generator's model distribution p_g.

Essentially, the Generator's function learns to map a data point sampled from this random distribution p_z to the target real data distribution p_x. Of course we can't expect that this mapping will be perfect, so we refer to as p_g, the new distribution created by the Generator function.

As previously explained, the primary goal of the Generator is to maximize the probability of creating data that the Discriminator classifies as real. This means that when the Discriminator receives a fake generated input, x_g, it should indicate that the input is from the real data distribution. In mathematical terms, this can be expressed as $D(x_g) = 1$.

In his research, Goodfellow formalized this idea by proposing the minimization of the Generator's objective function:

$$V(G) = E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.2}$$

Note that minimizing this objective function is the same as maximizing:

$$V'(G) = E_{z \sim p_z(z)}[\log(D(G(z)))] \tag{2.3}$$

The first function states that the Generator wants the minimize the amount of times the Discriminator is right and, in the other, it wants to maximize the times it is wrong.

[3] states that the minimization problem tends to saturate, since the initial generated outputs are clearly different from the real and that solving the maximization problem provides stronger gradients early in training.

2.5 Adversarial Training of the Discriminator

The Discriminator is another neural network represented by the function $D(x; \theta_d)$, where θ_d are learnable parameters.

The Discriminator receives data x as input and outputs a corresponding probability value $y = D(x)$ indicating the likelihood of that input data being real by assigning a value closer to 1 or fake, assigning a value closer to 0.

The Discriminator has two main objectives. It wants to maximize the times it correctly says a real image is indeed real. It also wants to maximize the times it correctly says a fake image is indeed fake. In mathematical terms, this is expressed as $D(x_r) = 1$ and $D(G(z)) = 0$.

In his research, Goodfellow formalized this idea by proposing the maximization of the Discriminator's objective function:

$$V(D) = E_{x \sim p_{data}(x)}[\log(D(x))] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.4}$$

2.6 Adversarial Training

The adversarial minimax game can finally be defined as:

$$\min_G \max_D V(G, D) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.5}$$

following the proposed training algorithm:

Algorithm 1 GAN training algorithm

```
for number of training iterations do
    1. Update Discriminator
        • Create a batch of $m$ noise samples $\{z^1, ..., z^m\}$
        • Create a batch of $m$ examples $\{x^1, ..., x^m\}$ of real data
        • Update the Discriminator by ascending it's stochastic gradient
              $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log(D(x^i)) + \log(1 - D(G(z^i)))]$    (2.6)
    2. Update Generator
        • Create a batch of $m$ noise samples $\{z^1, ..., z^m\}$
        • Update the Generator by descending it's stochastic gradient
              $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^i)))]$    (2.7)
end for
```

The intuition behind the training algorithm is that we begin by updating the Discriminator's weights so it has a preliminary knowledge of distinguishing real and fake examples.

After that, we update the Generator by using the Discriminator's gradient as guide. By using the gradient of the Discriminator with respect to fake samples, we can update the Generator to create examples that better represent real data.

2.7 GAN Loss

Implementations of the GAN framework use the Binary Cross Entropy Loss function during training.

2.7.1 Entropy

Entropy [14] measures the degree of uncertainty of a given distribution p:

$$H(p) = \sum_x p(x) \cdot log(\frac{1}{p(x)}) = -\sum_x p(x) \cdot log(p(x)) \tag{2.8}$$

For example, imagine we have 2 events x_0 and x_1, and their observed probabilities are $p(x_0) = 0$ and $p(x_1) = 1$. By inputting it into Equation 2.8 we get that the entropy is 0. This means that we are certain that event x_1 will happen, since it's probability is 100%.

Alternatively, if $p(x_0) = 0.5$ and $p(x_1) = 0.5$, we'd get that the entropy is log(2), indicating a higher degree of uncertainty. As the number of possible outcomes grows or, more importantly, as the probabilities become more indistinguishable and closer together, the larger the entropy will be.

2.7.2 Cross Entropy

The cross entropy [14] emerges when we want to approximate an unknown distribution p by some other distribution q:

$$H_p(q) = -\sum_c q(y_c) \cdot log(p(y_c)) \tag{2.9}$$

If both distributions are equal, the cross-entropy and entropy will be the same. But if they are not, the difference between them tells us how similar, or not, they are. This difference is refered to as Kullback-Leibler Divergence [14]:

$$D_{KL}(q||p) = H_p(q) - H(q) \tag{2.10}$$

The closer the distribution p get to the know distribution q, the lower their divergence will be.

2.7.3 Binary Cross Entropy

The Binary Cross Entropy is a derived form of the general Cross Entropy equation when there are only two output classes involved:

$$-\frac{1}{N} \cdot \sum_{i}^{N} [y_i \cdot \log p(y_i) + (1 - y_i) \cdot \log(1 - p(y_i))] \tag{2.11}$$

Each y_i output can take a value of either 1 or 0. If $y = 0$, the first part of Equation 2.11 goes away and when $y = 1$, the second part goes away.

2.7.4 BCE and GANs

By observing Equation 2.11 we can clearly see their relation with the GAN objective function, Equation 2.5.

When we pass a $y_i = 1$, real data, we are calculating the probability of data being real, or $\log(D(x))$. Similarly, when we pass a $y_i = 0$, false data, we are calculating the probability of data having been generated, or $\log(1 - D(G(z)))$.

The average over the summation of each y_i input returns the expectation of each value, as intended by the original value function.

2.7.5 Initial results

The original paper [3] offered interesting results. Note that for each of the following samples, the rightmost column shows the nearest training example of the neighboring generated sample. Figure 2.6 illustrates some of the outputs generated by the model when tasked with creating digits. Beyond that, Figure 2.7 demonstrates the potential of GANs in generating realistic-looking faces. Finally, Figure 2.8 shows RGB image samples of random objects.

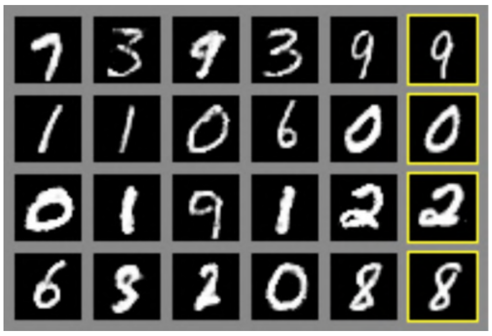


Figure 2.6: GANS trained on MNIST, from [3].

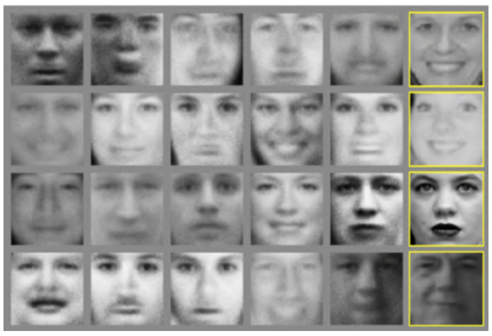


Figure 2.7: GANS trained on TFD, from [3].

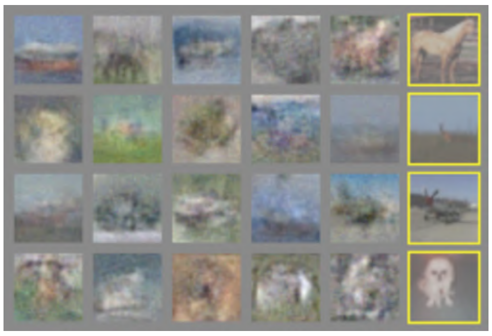


Figure 2.8: GANS trained on CIFAR-10, from [3].

These initials results highlight the potential of the adversarial framework for generating images, while also revealing its current limitations in interpreting RGB images. However, future research address this issues and manage to propose solutions to overcome them.

2.8 Conditioning GANs

The original model proposed by Goodfellow *et. al*, had promising results which were quickly improved upon by the research community. One such improvement involved addressing the issue of excessive randomness in generated outputs.

To understand the problem, consider a Generator trained on the MNIST dataset. This dataset is commonly used in AI research and consists of a large database of hand written numbers ranging from zero to nine. A generator network trained on this dataset has the ability to output a random image of one of these digits. However, there are situations where the desired outcome is not just any number, but a specific one. Unfortunately, unconditional generators offer no control over the output. In order to tackle this challenge, researchers delved into the topic of conditional generation, as discussed by Mirza *et. al* in [4].

The original unconditional model can be extended to a conditioned model by appending extra information y to both generator and discriminator nets:

$$\min_G \max_D V(G,D) = E_{x\sim p_{data}(x)}[\log D(x|y)] + E_{z\sim p_z(z)}[\log(1 - D(G(z|y)))] \qquad (2.12)$$

In common implementations of this network, the extra information is incorporated directly into the input layers. Specifically, the Generator receives this extra information as a one-hot vector label which is appended to the random noise vector and the discriminator as a one-hot matrix label which is added as extra image channels.

A one-hot vector is simply an array consisting of zeroes and ones portraying a specif label among a predetermined set. For example, the array $\{0,0,1,0\}$ will represent one possible class out of other four possible different ones. Similarly, a one-hot matrix follows the same idea, but instead of an array, it uses an entire grid of ones or zeroes to represent its labels, for example: $\{\{0...\},\{0...\},\{1...\},\{0...\}\}$. The idea of appending an extra information y to both discriminator (x) and generator (z) inputs is illustrated in Figure 2.9.

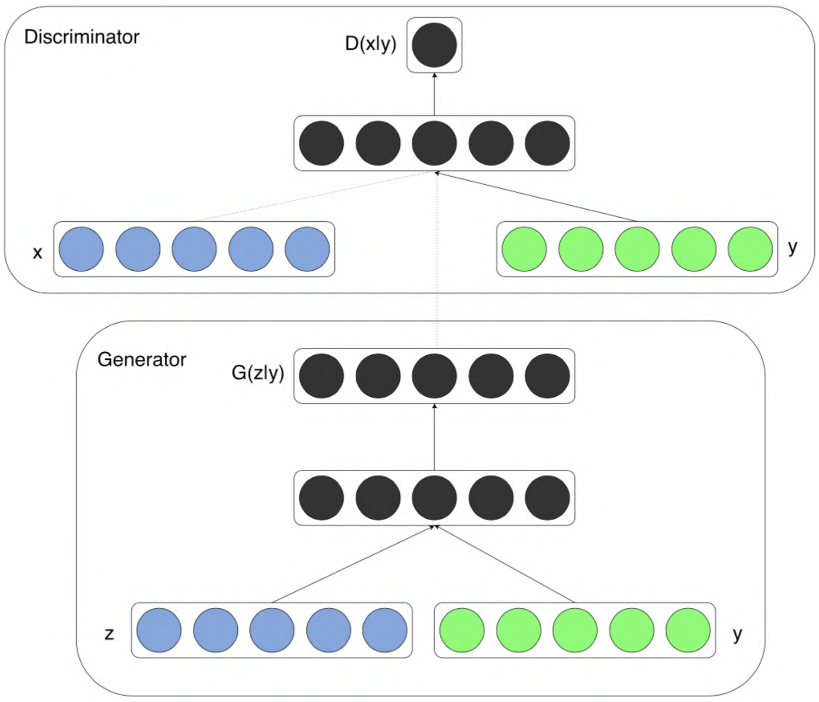


Figure 2.9: Appending extra information as input to network, from [4]

While this approach serves well for explanatory purposes, newer implementations have shifted towards using embedding layers in order to optimize input size.

Now, since both generator and discriminator nets are provided with the same class label, the generator will not only be trained on the discriminator's ability to distinguish between fake or real data but also on its ability to distinguish real or fake representations of a particular class.

Once you reach the production stage, modifying the one-hot vector y allows the generation of class-specific outputs.

2.9 Issues with GANs

Despite the great success in GANs, instability during training led to issues such as mode collapse and vanishing gradients. The study conducted by [15] explores the training dynamics of generative adversarial networks and makes the initial steps towards understanding these problems.

Mode Collapse A common problem faced by generative models is known as Mode Collapse, where a model fails to generate diverse samples and is reinforced into creating a limited range of outputs. An example of this would be if a GANs network trained on the MNIST digits dataset would be stuck into generating only 0s or 9s, failing to generate the other classes.

Vanishing Gradients Another common problem faced by GANs is called vanishing gradients. This problem arises from the utilization of the BCE loss function to model the minimax game. The Discriminator, having a relatively easier task, advances faster than the Generator which needs to simulate complex features, specially at the beginning of training. When the Discriminator is optimized too fast, its gradient tends to zero and the update it gives to the Generator gets worse.

2.10 Convolutional GANs

One notable contribution that addressed the instability of GANs during training was explored by Radford *et. al* [16]. In their research, they propose a set of rules that improve instability.

The authors explore the use of Convolutional Networks to upscale images. They explain that other attempts at implementing this architecture had been unsuccessful, but through extensive exploration they achieved success.

This success was mainly due to the fact that they adopted a set of guide lines to their CNN architecture:

- Replace pooling layers with convolutional layers.

- Use of batchnorm.
- Remove fully connected layers for deep networks.
- Use ReLU activation layer for all layers of the Generator, except for the last, which uses the Tanh activation.
- Use LeakyReLU for all layers of the Discriminator.

By adopting these guidelines, [16] observed notable improvements in training stability. The authors explain that batch normalization of the input helped in preventing the Generator from collapsing. Furthermore, they also explain that replacing pooling layers for strided convolutions allowed the model to learn its own spatial sampling. Lastly, they explain that the use of bounded activation functions allowed the Generator to learn more quickly.

Despite this architecture being able to generate pleasing results, it still lacked theory on explaining its stability, relying mainly on heuristic findings.

2.11 Wasserstein GANs

Another improvement came from identifying problems of the original proposed BCE loss function, which led to the introduction of a new method for calculating the loss term, as proposed in [17].

2.11.1 Problem with BCE

As explained by the authors, BCE essentially minimizes the KL divergence between the real data distribution p_r and the generated distribution p_g. However, there are situations where these distributions are too far apart and have no type of intersection. In such cases, the KL distance tends to become undefined and impractical for measuring the discrepancy between distributions.

One proposed solution is to add noise to generated images, in order to artificially shift the distribution towards the objective data distribution. This noise, however, degrades the quality of the outputs, making them blurry. The authors of [17] state that *the added noise term is clearly incorrect for the problem.*

2.11.2 Earth Movers Distance

Opposing KL divergence, [17] propose using another method, knwon as Earth Movers Distance (EMD):

$$W(p_r, p_g) = \inf_{\gamma \epsilon \Pi(p_r, p_g)} E_{(x,y)\sim\gamma}[\|x - y\|] \tag{2.13}$$

to calculate the difference between probabilities.

Intuitively, EMD calculates the amount of effort required to transform one set or distribution into another. This is measured by calculating all multiple plans (γ) for transporting data from the source distribution to the target and finding the one with minimal cost. In mathematics, this operation is represented by the infimum (inf) operator.

The authors state that calculating every transport plan is impractical, however present that EMD can be approximated using the Kantorovich-Rubinstein duality [18], resulting in:

$$W(p_r, p_g) = \sup_{\|f\|_L \leq 1} E_{x\sim p_r}[f(x)] - E_{x\sim p_g}[f(x)] \tag{2.14}$$

where $\|f\|_L \leq 1$ means that the function f is 1Lipschitz continuous, has a slope less than or equal to 1. The supremum (sup) defines the smallest value of a set that is larger than all values of a given subset. An intuition of the duality between infimum and supremum is depicted in Figure 2.10, where the filled green balls represent the subset S of the ordered set P of balls.

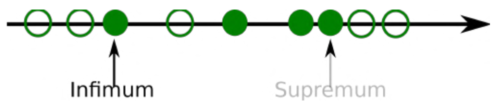


Figure 2.10: Infimum vs Supremum, from `wikipedia.com`

This meant that by bounding the probability functions of the GANs architecture the objective functions is simplified to a straightforward subtraction. Beyond that, one could train the Discriminator to an optimal state and still receive informative gradients to improve the Generator. Consequently, this resulted in a more stable training and reduction of mode collapse.

2.11.3 WGAN Loss

The objective function for WGAN obtained by the Kantorovich-Rubinstein duality is expressed as:

$$\min_G \max_D V(G, D) = E_{x_r\sim p_r}[D(x_r)] - E_{z\sim p_z}[D(G(z))] \tag{2.15}$$

It is important to notice that in this setup the Discriminator is no longer functioning as a classifiers. Instead, it is called a *critic*, where values obtained from the loss aren't restricted to the range of 0 and 1.

2.11.4 Enforcing K-Lipschitz Continuity

Initially, to ensure that the gradients of the network remained bounded, the authors of [17] proposed manually clipping the weight of the Discriminator to lie within a compact space $[-c, c]$.

However, as stated by them *Weight clipping is a clearly terrible way to enforce a Lipschitz constraint.* The first reason for this is that choosing a boundary for weight clipping involves empirical results, which makes it difficult to find a general solution. Secondly, by clipping the weights of the Discriminator we limit it's ability to effectively learn.

2.11.5 Improving the Enforcement of K-Lipschitz Continuity

It didn't take long for new approaches of enforcing K-Lipschitz continuity to arrise. One such approach, proposed in [19], introduces a method of penalizing the gradient of the Discriminator.

The authors first state that a function is 1-Lipschtiz if and only if it has gradients with norm at most one. Based on this statement, instead of directly clipping the weights, they introduce a regularization term that penalizes the Discriminator's gradient when it exceeds a norm of one. A λ term is added to control the magnitude of this penalty.

$$\begin{aligned}\min_G \max_D V(G, D) = E_{x_r \sim p_r}[D(x_r)] - E_{z \sim p_z}[D(G(z))] \\ +\lambda E_{\hat{x} \sim p_{\hat{x}}}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]\end{aligned} \tag{2.16}$$

From the WGAN-GP loss function, Equation 2.16, we can observe that the penalization term acts over the Discriminator's gradient over a new $p_{\hat{x}}$ distribution. This distribution is created by interpolating real and fake images by a random fraction ϵ, as explained in Algorithm 2.

It is worth noting that penalizing the gradient does not enforce Lipschitz Continuity, rather encourages it.

Algorithm 2 Interpolating Real and Fake Images

```
$x_g \leftarrow G(z)$
$x_r \leftarrow p_r$
$\hat{x} \leftarrow \epsilon x_r + (1 - \epsilon) x_g$
```

Although the results were promising, penalizing the gradient comes with a severe computational cost and, because of that, a new method of constraining the norm of the gradient was proposed.

2.11.6 Spectral Normalization

Takeru Miyato *et. al* [20] propose a new method called Spectral Normalization to ensure Lipschitz continuity. In their research, the team demonstrates how this method is both simple and computationally efficient, offering a significant advantage over previous approaches.

The core idea of spectral normalization is to controll the Lipschitz constant by constraining the spectral norm of each layer in the Discriminator network. The spectral norm of a matrix A is defined as:

$$\sigma(A) := \max_x \|Ax\|_2 : \|x\|_2 \leq 1 \tag{2.17}$$

and is equivalent to the largest singular value of matrix A. The authors use the inequality $\|g_1 \cdot g_2\|_k \leq \|g_1\|_k \cdot \|g_2\|_k$ to establish the following proposition:

$$\|f\|_k \leq \prod_{l=1}^{L+1} \sigma(W^l) \tag{2.18}$$

which basically means that the network is k-Lipschitz bounded by the product of the norms of each layer.

The proposed method aims to mantain the norm close to one, $\sigma(W) = 1$, by dividing the weight matrix over it's norm:

$$W_{sn} := \frac{W}{\sigma(W)} \tag{2.19}$$

By applying this normalization to every layer l of the discriminator network, the authors ensure a 1-Lipschitz bound on the discriminator function.

2.12 Image Translation

Image-To-Image Translation is a common type of computer vision and machine learning task that involves generating a new image from an input image while preserving certain visual characteristics of the original [21].

Some tasks that exemplify image-to-image translation include:

- Style Transfer: A task that involves applying an artists technique or style to a brand new image. Style transfer allows us to see how a random picture might have looked like if it were created by a famous painter;

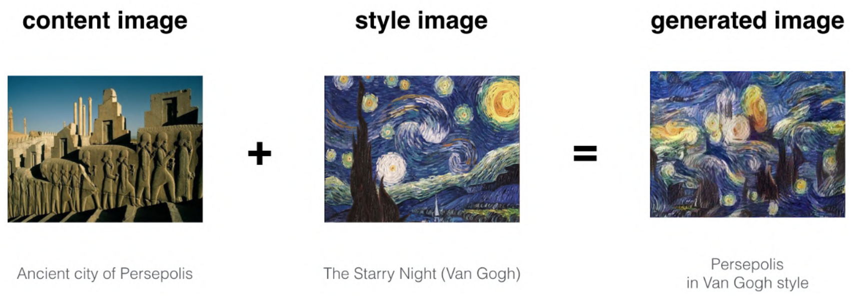


Figure 2.11: Example of style transfer, from `towardsdatascience.com`

- Colorization: A task that involves changing the color of images, such as adding color to black and white images, or transforming a light themed web page to a dark theme;

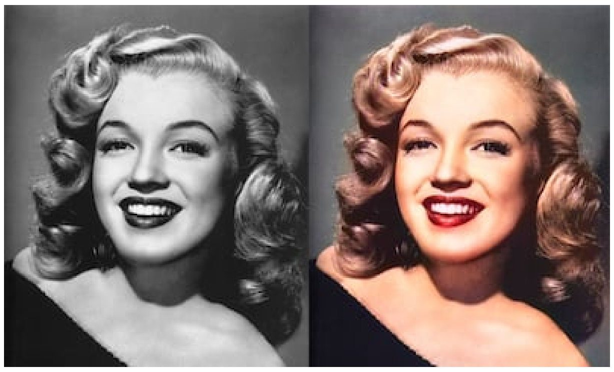


Figure 2.12: Example of colorization, from `hotpot.ai`

- Image Inpainting: A task that involves reconstructing missing parts of an image or removing undesired ones. This is particularly useful for restoring parts of an image or photo editing.

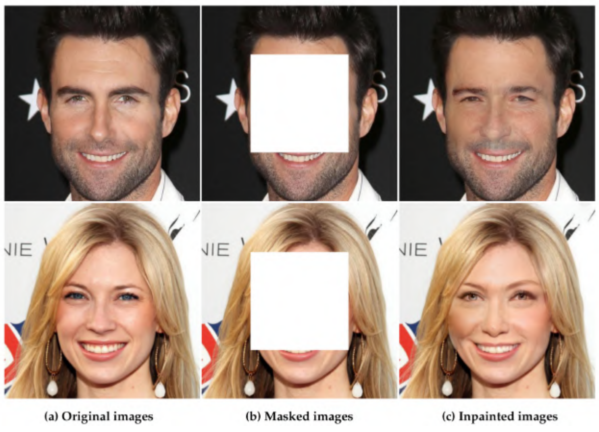


Figure 2.13: Example of image inpainting, from `mdpi.com`

The goal of image to image translation is to enable machines to learn how to transform images in ways that are visually pleasing and useful for a variety of applications.

2.13 Pix2Pix

Isola *et. al* in [22] introduce a new GANs architecture called Pix2Pix that proposes a solution to the task of general purpose translation of images from one representation to another.

In their research they explore and improve upon the techniques previously mentioned in order to present a simple framework that can achieve sufficiently good results.

To build their Generator and Discriminator networks, the Pix2Pix research team took inspiration on the Deep Convolutional GANs architecture with a few twists.

2.13.1 U-Net Generator

State-of-the-art solutions to the translation problem often employ an encode-decoder structure [5]. This structure consists of a series of layers in which the input is passed down through. The encoder component progressively down-samples the input, reducing its spatial dimensions until it reaches a bottleneck layer. It is interesting to note that this process maps the input to a representation of the latent z space.

After the bottleneck, the decoder component follows the inverse process, up-sampling the data back to its original size.

The purpose of down-sampling and up-sampling is to allow the network to capture and learn the most important features of the input. However, the Pix2Pix team observed that this process discards desirable low-level features.

To address this issue, they included skip connections and follow the shape of a "U-Net", in which each down-sample layer is connected to its up-sample layer conjugate. Each skip connection concatenates all channels from one layer to the other, allowing the network to capture both high-level and low-level features of the input.

2.13.2 Pixel Loss

Another improvement the Pix2Pix team explored focused on the benefits of incorporating an L1 loss term alongside the original GANs objective function. This approach was inspired by previous studies that demonstrated that including the L1 loss term encouraged the Generator's output to be closer to the input. While previous research used L2 distance to calculate the loss, the Pix2Pix team found that the L1 loss encouraged less blurring.

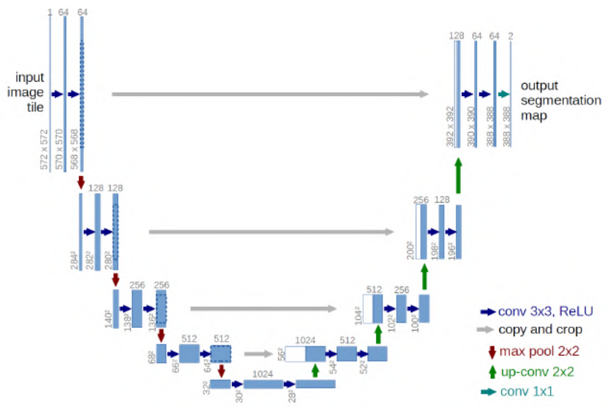


Figure 2.14: U-Net Architecture, from [5]

The L1 loss measures the absolute pixel-wise difference between the fake generated output and the target data. By adding this loss term, the Generator is able to capture low frequency discrepancies between the output and the target, ensuring that the Generator outputs data closer to target.

$$L_{L1}(G) = E_{x,y,z}[\|x_r - G(z, y)\|] \tag{2.20}$$

2.13.3 PatchGAN Discriminator

Since the Generator already has the ability to measure low frequency errors between output and target, the Pix2Pix team noted that they could restrict the Discriminator to only observe the high frequencies. To accomplish this, they designed a Discriminator architecture called PatchGAN.

Unlike previous Discriminators, that analyzed the image as a whole, the PatchGAN convolutes across the image and analyzes smaller NxN patches of the image individually, classifying it as real or fake. The final Discriminator's output loss is the averaged sum of all patches.

By analyzing small patches of the image, the Discriminator concentrates on the high-frequency features. This process proved to be faster, since the Network turned out to be smaller with fewer parameters, and returned better feedback to the Generator, even when applied to arbitrarily large images.

2.13.4 Pix2Pix Architecture

The appendix of [22] provides comprehensive details of the Pix2Pix network architecture and parameters. The final architecture and objective function, Equation 2.21, combine aspects from Conditional GANs, WGANs and Deep Convolutional GANs.

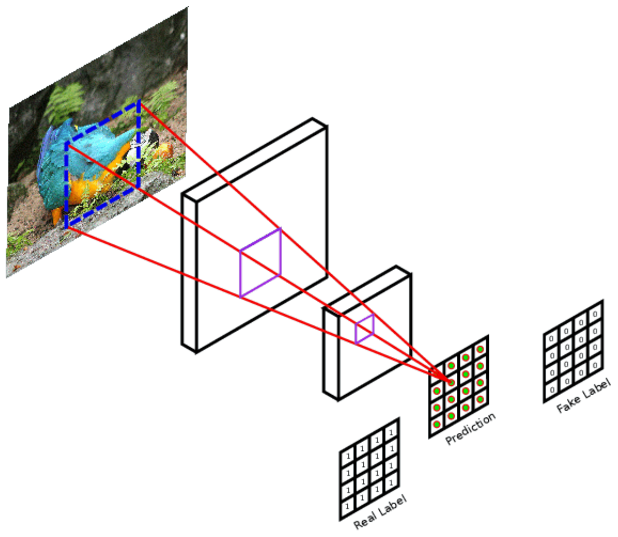


Figure 2.15: PatchGAN discriminator classifying a patch of the image, from `researchgate.net`

$$\min_G \max_D V(G,D) = E_{x_r \sim p_r}[\log D(x_r, y)] - E_{z \sim p_z}[\log(1 - D(G(z,y),y))] \\ + \lambda E_{x_r, z \sim p_r, p_z}[\|x_r - G(z,y)\|] \tag{2.21}$$

The U-Net Generator follows a encoder-decoder architecture. The encoder portion of the network, down-samples an input 256x256 RGB image to a 512 feature vector. Each down-sampling step uses **Convolutional** layers to down-sample the given input. Each convolution is followed by **BatchNormalization** and **LeakyRelu** activation layers.

The decoder portion of the network up-samples the feature vector back to a 256x256 RGB image. Each up-sampling step uses **TransposedConvolutional** layers to up-sample the given input. Each transposed convolution is followed by **BatchNormalization** and **Relu** activation layers. The final layer consists of a **Tanh** activation to guarantee that the final output is within a normalized range of [-1,1].

The encoder and decoder portions of the network are connected via skip connection, that concatenate the output of the decoder to the corresponding input of the decoder. As mentioned previously, this helps preserving finer details that can be lost during down-sampling.

Batch normalization re-scales the output of a layer to have 0 mean and standard deviation of 1, over a batch sample. A specific normalization called **InstanceNormalization** occurs when the batch size is 1, resulting in the normalization of the image within its own pixels. Activation functions, as the name suggests, decides whether or not a neuron should be activated. It's main purpose is to add non-linearity to the network in order to simulate complex functions. As discussed in Section 2.10, applying these techniques makes training of the model faster and more stable.

The PatchGan Discriminator used in Pix2Pix outputs a 30x30 probability grid indicating whether a corresponding 70x70 patch of pixels is real or fake. During training, it receives the concatenated pair of real and fake images, as input, and uses **Convolutional** layers followed by **BatchNormalization** and **Relu** layers to output the desired probability matrix.

Each training loop starts by generating a fake sample. Then, starting with the discriminator, both networks are alternately updated. The discriminator employs **Binary Cross-Entropy** (BCE) loss to ensure that it correctly identifies generated samples as fake and valid samples as real. Contrary, the generator employs **BCE** loss to encourage the discriminator to classify generated samples as real. Additionally, the generator employs **L1** loss between the fake generated output and the desired output to encourage the generation to create images more closely related to the desired outcome.

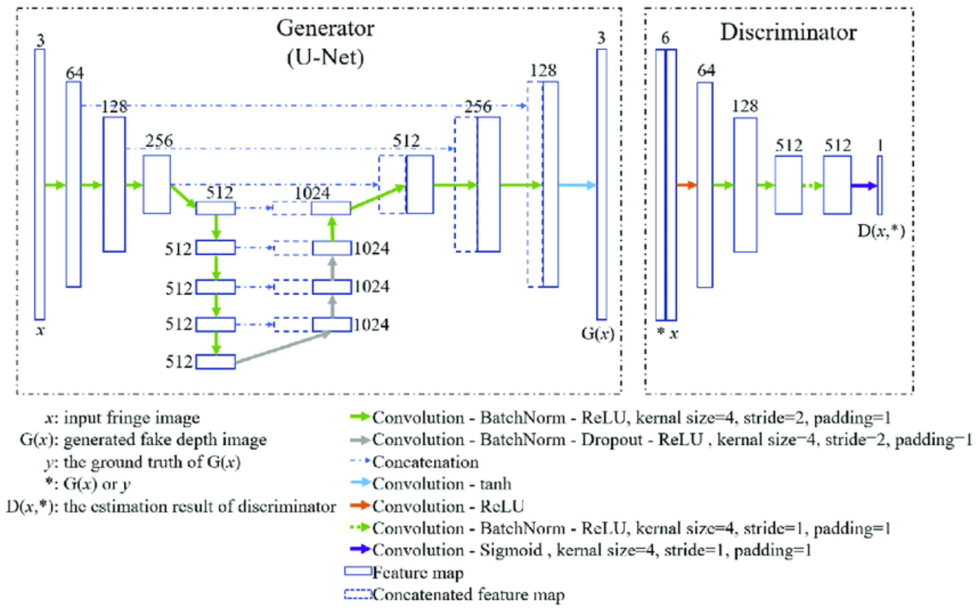

Generator
(U-Net)
Discriminator
3
64
128
256
512
1024
512
1024
3
x
G(x)
6
512
1
D(x,*)
* x
x: input fringe image
G(x): generated fake depth image
y: the ground truth of G(x)
*: G(x) or y
D(x,*): the estimation result of discriminator
Convolution - BatchNorm - ReLU, kernal size=4, stride=2, padding=1
Convolution - BatchNorm - Dropout - ReLU , kernal size=4, stride=2, padding=1
Concatenation
Convolution - tanh
Convolution - ReLU
Convolution - BatchNorm - ReLU, kernal size=4, stride=1, padding=1
Convolution - Sigmoid , kernal size=4, stride=1, padding=1
Feature map
Concatenated feature map

Figure 2.16: Pix2Pix Architecture, from [6].

2.14 Evaluating GANs

The evaluation of generative models poses a unique challenge compared to traditional classifier networks. The usual technique for evaluating these kinds of neural networks consists of testing the model on a unfamiliar labeled dataset and verifying its accuracy in correctly classifying each label. However, the absence of concrete metrics that quantify the quality of generated data, makes it difficult to employ the same techniques used in classifier models to evaluate generative models.

With this in mind, proposing new evaluation methods is an active area of research where most used techniques revolve around assessing the quality and diversity of generate outputs. Measuring quality is crucial since we want the model to create realistic and high-quality data. In addition, measuring diversity of outputs is important, since a good generative model should not produce near identical samples repeatedly, even if each sample has high quality.

2.14.1 User Score

The most intuitive method for assessing image quality involves directly having human volunteers judge the visual quality of generated samples. This approach has obvious downsides. First, it requires finding and coordinating a large group of volunteers. Additionally, as explained in [23], outcomes vary depending on the test setup. The study observed that feedback from volunteers changed once they were made aware of their mistakes.

Considering this limitation, the research team recognized the need for a more general and automated evaluation approach. They proposed a novel method that involved using a classifier network to extract features of a image and calculate a final score of image quality.

2.14.2 Inception Score

Explained in [24], the Inception Score (IS) utilizes google's inception model, to create a single value that measures quality and realism of a generated image.

The inception classifier detects whether a image contains specific objects or not. More specifically, it takes a image as input and outputs a list of probabilities indicating the likelihood of each labels' presence in the image, summing up to one.

We can intuitively observe that if an image depicts only a single object, the classifier will output one big probability peak, the probability of that specific class, and many low probabilities to other known objects. However, if the image contains multiple objects, the classifier will output a list of multiple similar probabilities for each of the observed classes in the image. In other words, a image with a single object has low entropy, and images with more than one object have high entropy.

The research team proposes applying the Inception model to a large sample of generated images to obtain the conditional probability $p(y|x)$. If these images depict only a single object, which is the desired effect, they will present low entropy. Beyond that, in order to quantify the diverse aspect of images, the team defined that the marginal probability $p(y)$, the sum of all conditional probabilities, should have high entropy. The further apart these two distributions are from each other, the better the model is performing. The intuition for measuring the distance between distributions is depicted through the KL divergence:

$$IS = \exp(E_{x_g \sim p_g}[D_{KL}(p(y|x_g) \| p(y))]) \tag{2.22}$$

The final result is the exponent of the average distance for all images, where generated images will present clearly distinct labels and the overall set of images will represent diverse labels. The authors observed that this result closely correlated to evaluations made by human volunteers.

Despite being promising, the inception score presented two major drawbacks: first it was limited by the amount of classes known by the inception score. Second, the score can't correctly evaluate the model's ability to generate more than one image per class.

2.14.3 FID

The Frechet Inception Distance (FID), proposed in [25], poses itself as an improved metric that overcomes the limitations of the inception score.

Similar to its predecessor, the FID also utilizes the inception model, however, instead of extracting a probability vector, the FID leverages the network as a feature extractor to obtain feature embedding vectors from both generated and real images. These embeddings are essentially a condensed representation of each image.

The team's method consists of extracting the condensed information contained in intermediate layers of the inception network and model it as a Gaussian distribution with mean μ and covariance Σ. The final score can be calculated from the Wasserstein-2 distance between the modeled Gaussian distributions of generated and real image samples:

$$FID = \|\mu_r - \mu_g\|_2^2 + Tr(\Sigma_r + \Sigma_g - 2\sqrt{\Sigma_r \Sigma_g}) \tag{2.23}$$

,

where Tr is the **trace** operation which calculates the sum of all the elements in the matrix's diagonal.

2.15 Border Extraction

Border Extraction [26], also known as Edge Detection, is a common task in computer vision. It consists of segmenting an image by detecting areas of discontinuity in pixel intensity. Pixels that belong to regions with abrupt changes in intensity are called **Edge Pixels**. A set of connected edge pixels form an **Edge**. The main objective of edge detection is to develop algorithms that can detect these intensity changes and define boundaries of objects within an image.

Changes in intensity in an image can be detected using derivative operations. The derivative of an image is computed through spatial convolution with a filter kernel, usually a 3x3 matrix. The process of convolution consists of changing each processed pixel in the image with the sum of the products of the kernel weights and the corresponding pixel values within the region encompassed by the kernel. For example given a matrix

$$W = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} \tag{2.24}$$

and an image region

$$I = \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} \tag{2.25}$$

the resulting output for convolving W with I at pixel p_5 is

$$\hat{p}_5 = \sum_{k}^{9} w_k p_k \tag{2.26}$$

The process of convolution means applying this procedure through all pixels of the image.

2.15.1 Sobel and Laplacian Edge Detectors

Two commonly used kernels for computing spatial derivatives are the Laplacian and the Sobel kernels. The Sobel algorithm utilizes two kernels to compute the first-order derivative of an image in both horizontal (x) and vertical (y) directions. On the other hand, the Laplacian algorithm uses a single kernel to compute the second-order derivative.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.27}$$

(a) Horizontal Sobel Operator

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.28}$$

(b) Vertical Sobel Operator

Applying the first-order derivative produces thicker edges in points of constant intensity change, while the second-order derivative exhibits stronger response to finer details in

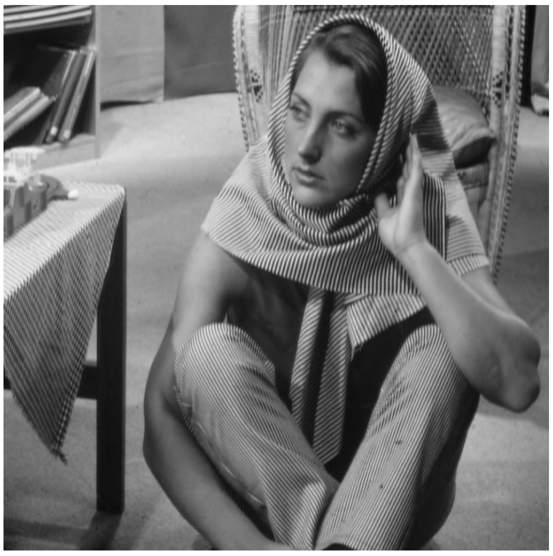
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.29}$$

Figure 2.18: Laplacian Operator

the image. Since the Laplacian kernel, used for calculating the second-derivative, is sensitive to noise, it is common practice to apply a Gaussian smoothing to the image prior to the convolution process, as explained by the Marr-Hildreth edge-detection algorithm [26].

2.15.2 Canny Edge Detector

Another commonly used algorithm for detecting edges is the Canny Edge Detector [26]. The Canny algorithm aims to improve upon the first derivative based detection by thinning the thick edges detected in the initial pass illustrated in Figures 2.19 and 2.20. This is achieved by a process called **Non-Maximum Suppression**, which consists of zeroing out the intermediate pixels that end up composing the thick edges of the image. Each pixel is compared with its two neighbors along the direction of the gradient and, if any of these neighbors has a larger pixel intensity, then the pixel is set to zero, otherwise, it stays the same. Following Non-Maximum Suppression, the image is further processed using threshold techniques to reduce false edges. The Canny algorithm improves upon normal thresholding by employing a technique called **Hysteresis Thresholding**, which uses two threshold value to classify each pixel in the image as either strong, weak or irrelevant. Irrelevant pixels are discarded, while strong pixels are assumed to be valid edge pixels. Weak pixels can be transformed into strong pixels through a process called Hysteresis, also known as Edge Tracking. If a weak pixel is 8-connected to a strong pixel, it is marked as a strong pixel. After process of hysteresis, the remaining weak pixels are also discarded, leaving only the strong pixels in the final result as illustrated in Figure 2.21.


(a) Original Image

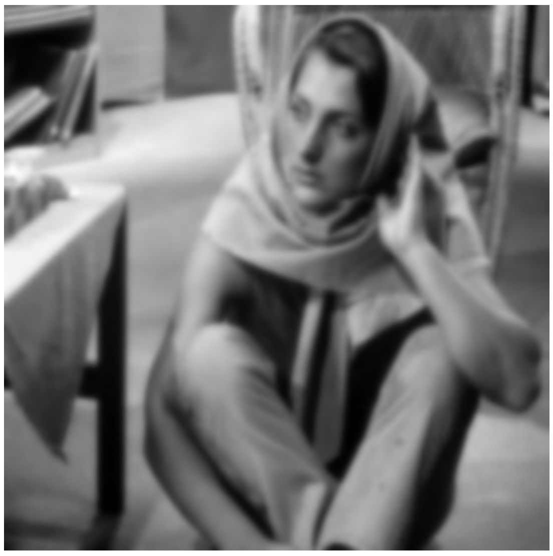
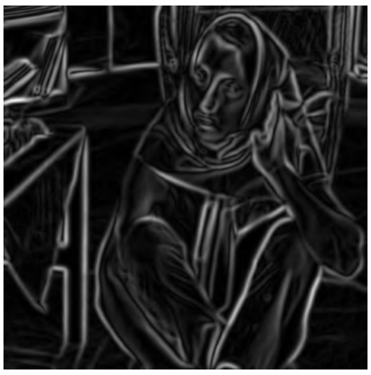
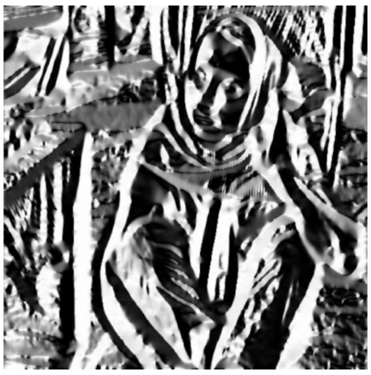

(b) Blurred Image

Figure 2.19: Result of applying gaussian blur to the original image


(a) Magnitude


(b) Gradient in X

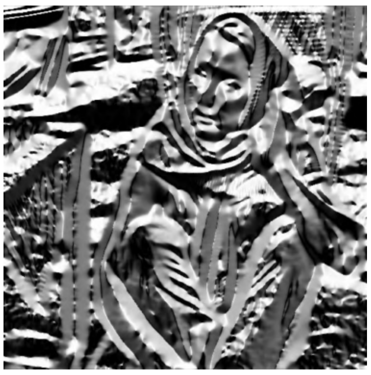
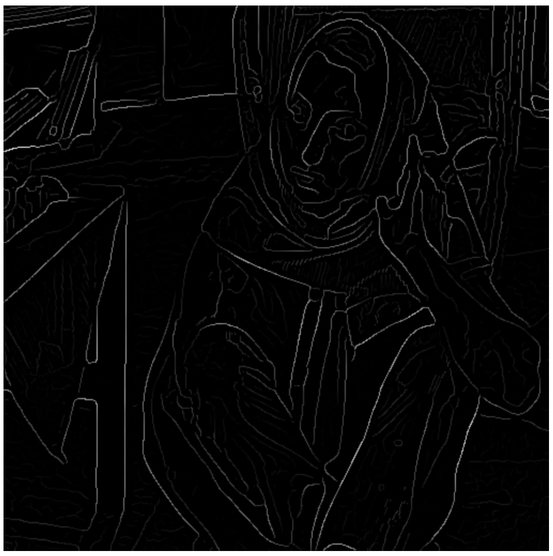

(c) Gradient in Y

Figure 2.20: Result of calculating the gradient of the blurred image


(a) Non-Maximum Suppression

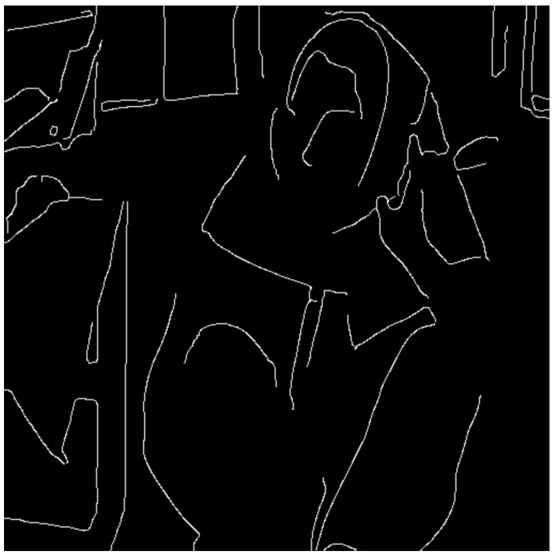

(b) Hysteresis Thresholding

Figure 2.21: Result of Applying Non-Maximum Suppression and Hysteresis Thresholding

Chapter 3

Development

In this chapter I will provide a detailed explanation of the problem I chose to tackle and the various solutions I developed to solve it. I will first highlight the significance of the problem and possible impacts it can make in the real world. After that, I will present the challenges I faced and the thought process behind each solution I came up with. Each solution presented its own additional set of problems that are addressed in subsequent sections.

A general overview of the experiments is:

- I begin by using the Pix2Pix model, pre-trained to transform Google Maps views into satellite images, to transform fantasy maps into satellite-like representations.
- Expanding upon the results of the first experiment, I employ border extraction techniques to create my own dataset which will be used to train the Pix2Pix model.
- With the model trained on this new dataset, I develop a drawing application that feeds user sketches to the network and instantly displays it's outputs.
- In the fourth experiment I create a new dataset consisting of scanned hand-drawn images of trees and rocks, without using border extraction techniques.
- With the intent of increasing the resolution of the model's output, I train the model with maps that contained high level of detail.

3.1 Translating Hand Drawn Maps

As an avid player of Dungeons and Dragons, a pen and paper board game that involves imagination, I often find myself assuming the role of Dungeon Master. As the Dungeon Master, one of my primary responsibilities is to come up with content and adventures for the players. An important aspect of this task involves crafting maps to represent the game world and give the players a visual perspective of the place they are in.

While the internet provides many tools that can be used to create good looking maps, most of them have a steep learning curve and can be time-consuming to master. As a result, most Dungeon Masters resort to drawing the map by hand. With this in mind, I came up with the idea of transforming these hand-drawn maps into ones that seemed to be created by the computer program, combining the versatility and ease of drawing the map by hand with the details of the ones generated with computer programs.

This idea led me to explore the concept of image translation. As I previously explained, image translation is a common computer vision task that can present itself in multiple forms. In my case, the task involved translating hand-drawn maps into computer-generated versions.

Although I presented the problem from the perspective of a Dungeons and Dragons game, it is important to note that this problem can be generalized to any other game or video game that involves map creation.

3.2 Translating Maps

I began my research by validating the idea of using the Pix2Pix model as a means of solving the task of transforming hand-drawn maps into refined computer-generated versions. To accomplish this, I needed two things: a working implementation of the model and an image of a RPG city map for testing. The diagram depicted in Figure 3.1 outlines this experiment's idea, where I submit a randomly generated RPG map to the Pix2Pix model and receive it's satellite representation.

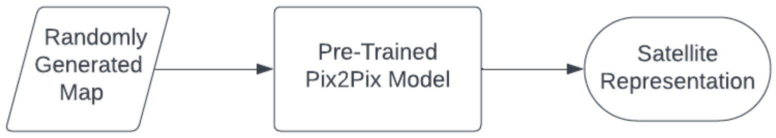


Figure 3.1: Diagram: Converting Random city map to Satellite

A PyTorch implementation of the Pix2Pix network is freely available on GitHub [2]. However, rather than using a untrained model, I opted to use a pre-trained version. This choice helped save up some time while still offering a convenient starting point for my experiments.

Regarding the RPG city map image, I decided to deviate from using a hand-drawn map. Instead, I used an online tool [7], to generate random images of RPG villages, which closely resembled the input used by the model during training.

This test provided important feedback regarding the suitability of the Pix2Pix model in this context. Although obvious adjustments were required to align the model with the original proposal, this initial validation proved that the model would be able to achieve the desired outcome.

3.2.1 Training Dataset

As I mentioned, rather than training the Pix2Pix model from scratch I decided on utilizing a pre-trained version of the model. The authors of Pix2Pix have made many pre-trained models available. After careful consideration, I decided that the **Map2Sat** pre-trained model would be the most suitable for my initial application. This particular model has been trained on the Maps Dataset, which consists of 1096 image pairs scraped from Google Maps. Each pair consists of the representation of a specific region in the world as seen in Google Maps view alongside its corresponding satellite view. This dataset had a close resemblance to the data I aimed to transform, making it great as a starting point. An illustrative example of this model's capabilities is represented in Figure 3.2, where the leftmost image represents the input, the image in the center is the expected output and the rightmost image is the generated output from the network.

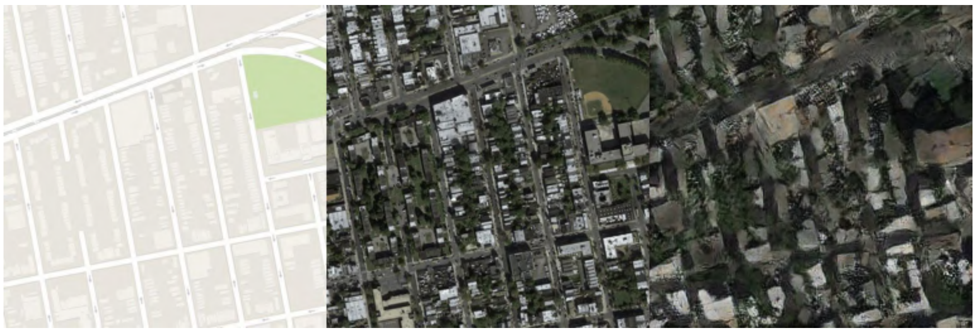
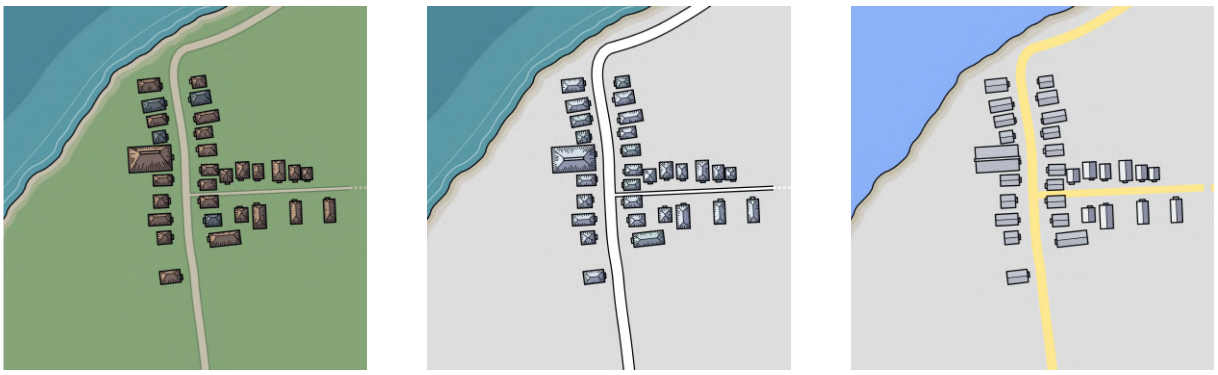


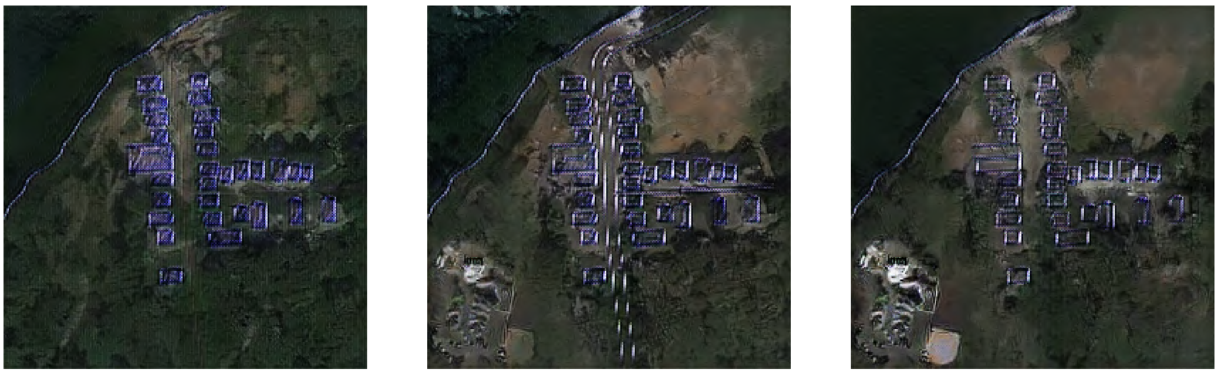
Figure 3.2: Map2Sat Input and Output Example.

3.2.2 Testing Data

The training data consists of a random village I created using the online tool mentioned previously. This tool provides the ability to customize the image's color palette. With this functionality, I attempted to create an image that would closely resemble the images in



(a) Color adjustments to the random input



(b) Generated outputs from corresponding adjustment

Figure 3.3: Inputs-Outputs pairs

the Maps Dataset. The images depicted in Figure 3.3a illustrate my process of adjusting the colors to better match the map images.

3.2.3 Outputs

The images in Figure 3.3b illustrate how each adjustment I made to the input image affected the output generated by the Pix2Pix model. We can see from the results that the model managed to understand what elements in the input image represent a desired generation of grass and water patches. However, it encountered difficulties in recognizing and representing the houses and roads, despite my best efforts to mimic their shape and color based on the training dataset.

3.2.4 Conclusions

Despite not achieving the desired results, this initial test provided encouraging insights opening up possibilities for improvement. Overall, it served two primary purposes: it provided an introduction to using the Pix2Pix network and it affirmed that the model has the potential to perform map translation tasks. Building upon these insights, I decided to modify the training approach. Instead of using a pre-trained model, I will create a custom dataset specifically tailored to the task of translating RPG maps.

3.3 Refining the Model

As I mentioned, after conducting my initial assessment of the Pix2Pix model, it became clear that I had to change the data used for training the model. After careful consideration, I decided to create a dataset specifically designed to meet the goals of my project. In the following sections, I will describe the reasoning behind my choice and the process for its creation. A general overview of this experiment is depicted in Figure 3.4, where a series of map images undergo edge detection, resulting in a dataset consisting of Edge-Map image pairs.

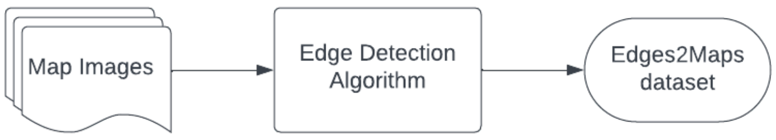


Figure 3.4: Diagram: Extracting borders from images to create Edge-Map dataset

3.3.1 Creating the Training Data

Initially, I utilized a model that was pre-trained on a dataset that didn't represent the desired end result. To address this, I decided to use a new dataset that would more closely represent it. While there are many RPG maps available on the internet, all of them are only the final image, without the intermediate sketches that got them to that point. Due to this lack of map sketches, I chose to simulate them by extracting the borders from the final map images.

Border Extraction, also known as Edge Detection, is a common computer vision task. In order to extract the borders of the map image, I explored three popular algorithms that detect edges: the Canny algorithm, the Sobel algorithm and the Laplacian algorithm. I applied each algorithm to a random RPG map image I got from the internet, Figure 3.5, and the results of each algorithm pass are depicted on the images of Figure 3.6.

As we can see the Sobel and Canny algorithms performed relatively well in detecting the edges, while the Laplacian algorithm resulted in excessive noise. Nevertheless, I observed that all resulting edge images contained too much detail and were not a faithful representation of a quick hand-drawn sketch. This outcome was mainly due to the input having too many details.

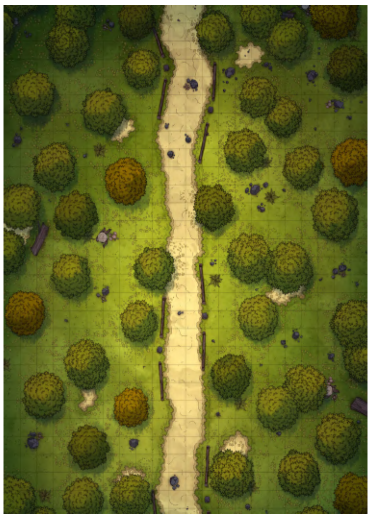
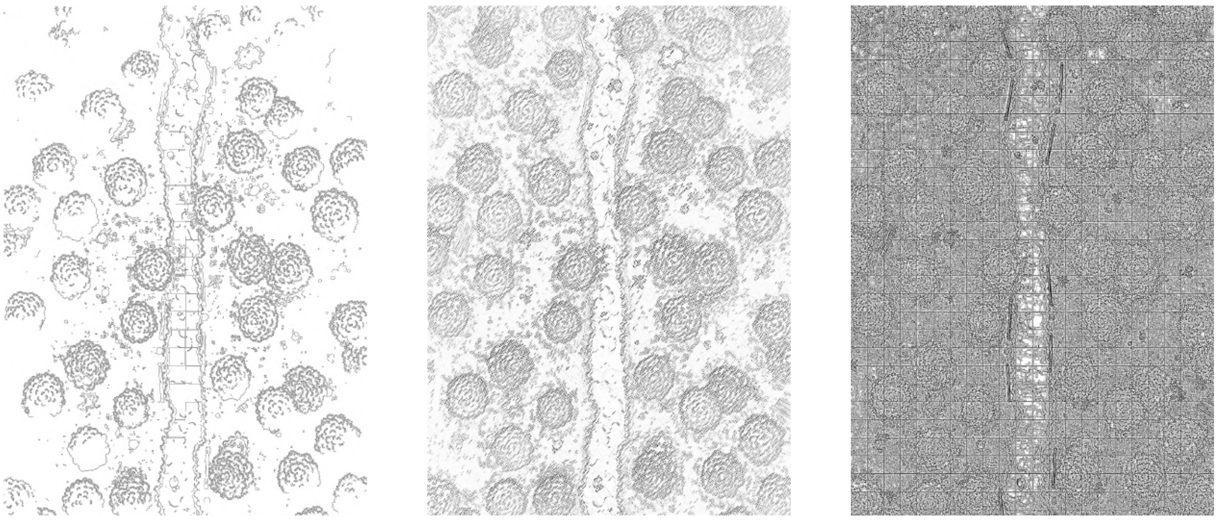


Figure 3.5: DnD map, from `reddit.com`.



(a) Canny (b) Sobel (c) Laplacian

Figure 3.6: Edge Detection Results: Canny, Sobel, and Laplacian Edge Detectors (High-Resolution Map)

To address this issue, I had the idea to continue using the village generator tool [7]. With this tool, I could create a map image with fewer details and extract the edges from it, resulting in an end result that would be more similar to a hand-drawn sketch. Again, I applied the same three algorithms to a new image, Figure 3.7, getting the corresponding results displayed on images in Figure 3.8. A visual evaluation, led to the Laplacian algorithm outperforming the other two algorithms, extracting all of the edges of the image. Consequently, it became my method of choice for creating the dataset.

In a python script, I applied the Laplacian algorithm on a variety of map assets I generated using the village generator tool. I concatenated each sketch output to the

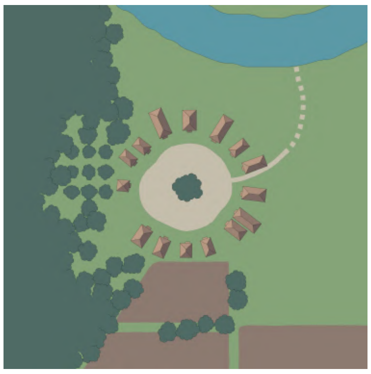
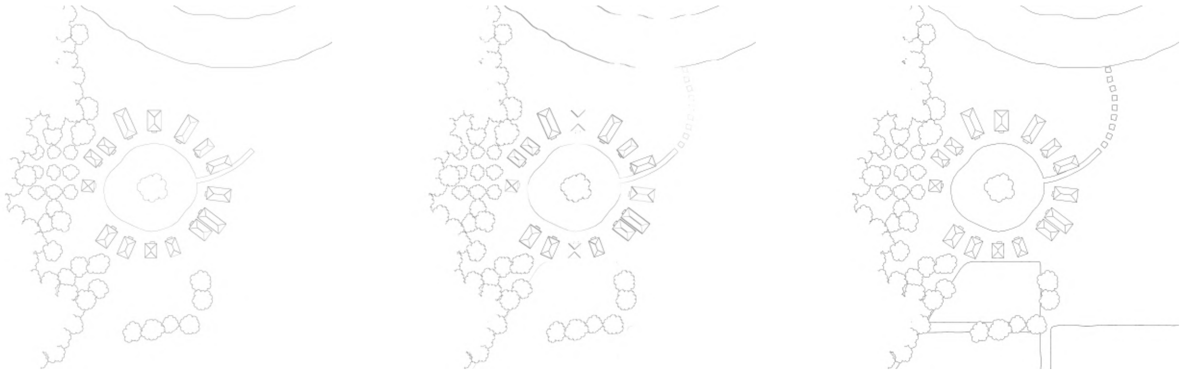


Figure 3.7: Map Randomly Generated with [7] .



(a) Canny (b) Sobel (c) Laplacian

Figure 3.8: Edge Detection Results: Canny, Sobel, and Laplacian Edge Detectors (Map with Fewer Details)

original image, resulting in the final improved dataset. Figure 3.9 illustrates an example of the concatenated image. After completing this process, the dataset consisted of 100 image pairs. Hopefully, training the Pix2Pix model with this dataset would enable it to generate map translations that aligned more closely with the desired outcome.

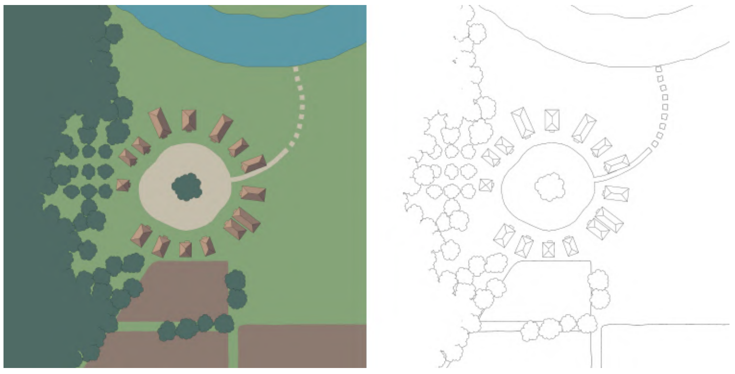


Figure 3.9: Example taken from dataset of the map and corresponding edges.

3.3.2 Training the Model with the New Dataset

Moving forward, the next step was to train the Pix2Pix model using the newly created dataset. For this training I used the same architecture as the Pix2Pix team [22] and to facilitate the process, I also used the script they offer for training the model from scratch using a custom dataset [2]. The script consists of a simple training loop ranging over 100 epochs. During each epoch, the model is trained with all the images in the dataset. An optional parameter called **batch_size** can be passed to the script in order to regulate the number of samples processed by the model before updating its parameters. I decided to experiment with different batch sizes and observe how each of them would affect the overall performance of the network.

3.3.3 Conclusions

Table 3.1 demonstrates the effect of varying the **batch_size** between 1, 5, 10 and 50 on the duration training the model for 100 epochs. We can see that increasing the batch size from 1 to 10 had a small effect on training duration. However, using a batch size of 50 significantly extended the time required to train the model.

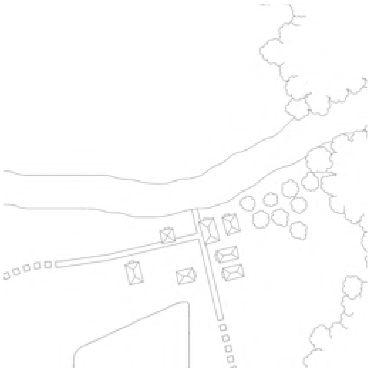
Table 3.1: Batch Size vs Duration

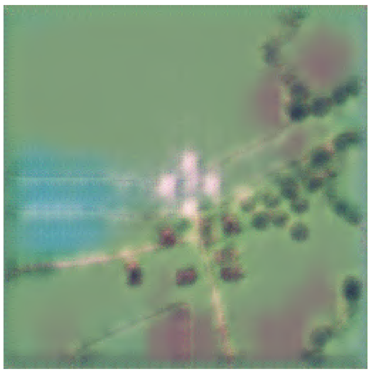
Batch Size	Training Duration
1	6159 seconds
5	6179 seconds
10	6190 seconds
50	6540 seconds

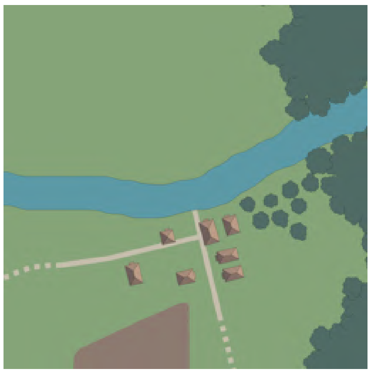
Just looking at the results from the table can be misleading, since only training duration is not the best metric for evaluating performance of the model. A better evaluation can be made by observing figures 3.10, 3.11, 3.12 and 3.13 that show outputs of the network for each batch size. Each figure respectively illustrate the input edges, the generated fake outputs and the desired real output images across epochs 4, 36, 68 and 100, demonstrating the evolution of the network after each training loop.

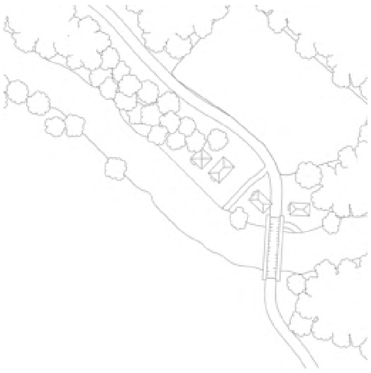
We can further observe Figure 3.14 and compare outputs after the network had been fully trained on 100 epochs. Based on these images, we can see that the networks trained with a **batch_size** of 1 and 5 generated images with a higher degree of similarity with the expected output, while the network trained with a **batch_size** of 50 generated blurred images. This is most likely due to the size of the dataset that didn't allow for enough updates of the network weights within each epoch.

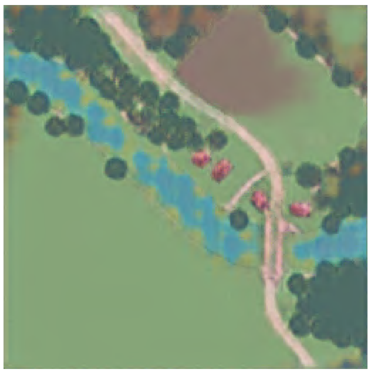
It is also interesting to note that the outputs obtained with **batch_size** of 5 and 10 demonstrated higher performance for generation of islands, where the entire body of water surrounding the main land is being accurately generated. In contrast, the output with **batch_size** of 1 generated an excessive amount of grass patches that should have been water. This phenomenon may arise from the fact that, during the training loop, the model can access more images before updating and therefore be less inclined to generate grass. It is also important to consider that the majority of images in the dataset do not depict islands, which could also have affected the output.

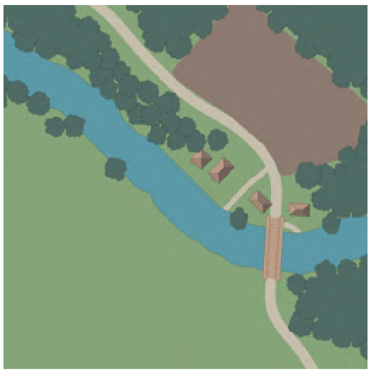

(a) 4-Epochs Input Edges

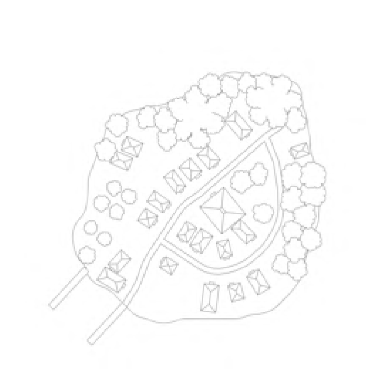

(b) Generated Output

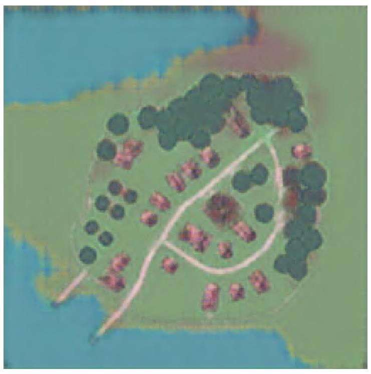

(c) Expected Output

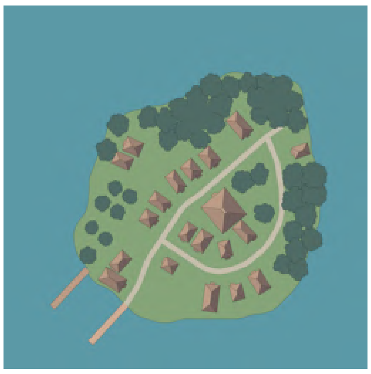

(d) 36-Epochs Input Edges

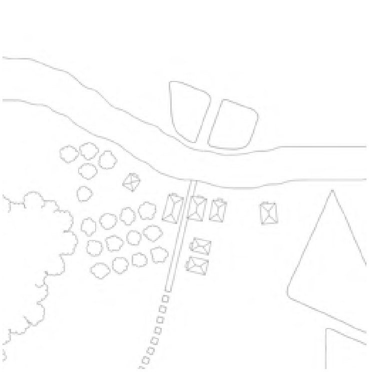

(e) Generated Output

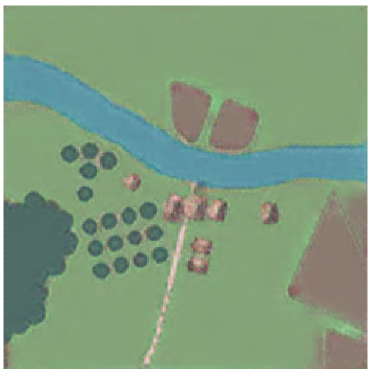

(f) Expected Output


(g) 68-Epochs Input Edges


(h) Generated Output


(i) Expected Output


(j) 100-Epochs Input Edges


(k) Generated Output

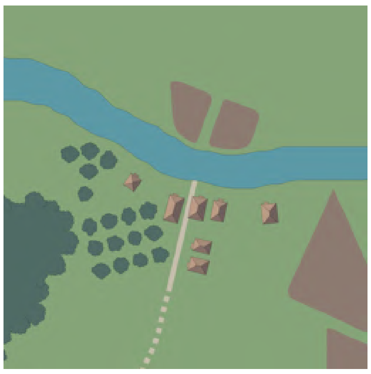
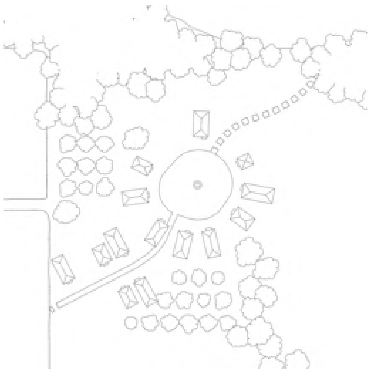
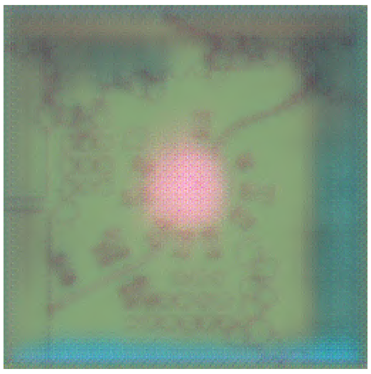
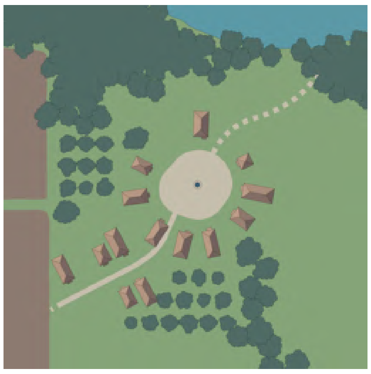
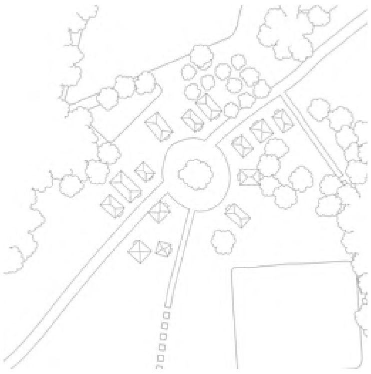

(l) Expected Output

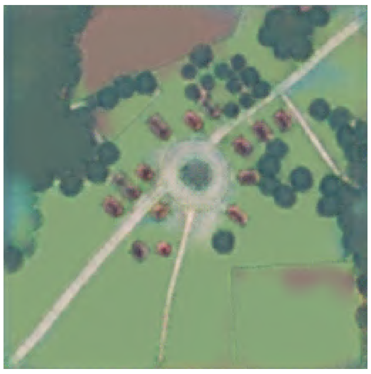
Figure 3.10: Training progress with Batch Size of 1 after each 32 epochs, starting from 4

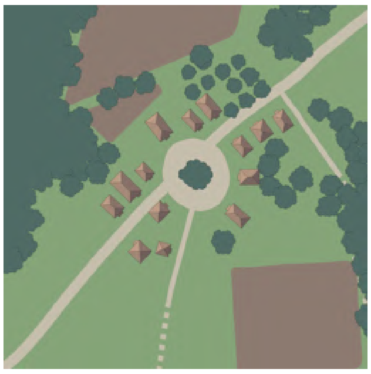

(a) 4-Epochs Input Edges

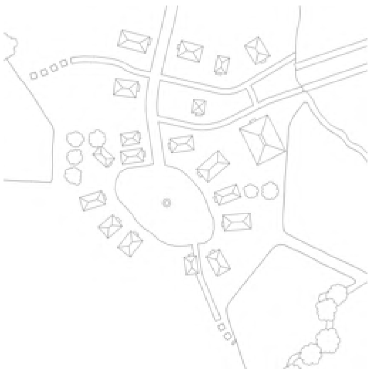

(b) Generated Output

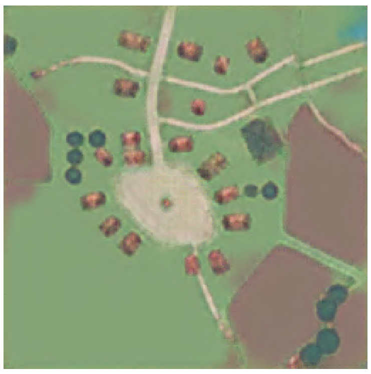

(c) Expected Output

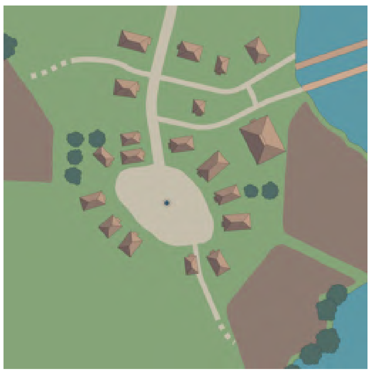

(d) 36-Epochs Input Edges

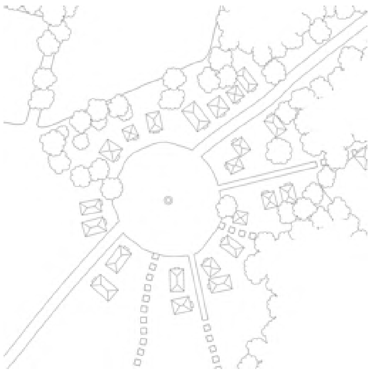

(e) Generated Output

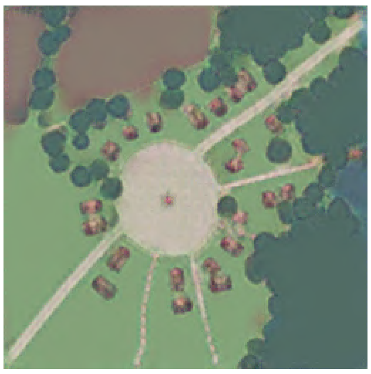

(f) Expected Output


(g) 68-Epochs Input Edges


(h) Generated Output


(i) Expected Output


(j) 100-Epochs Input Edges


(k) Generated Output

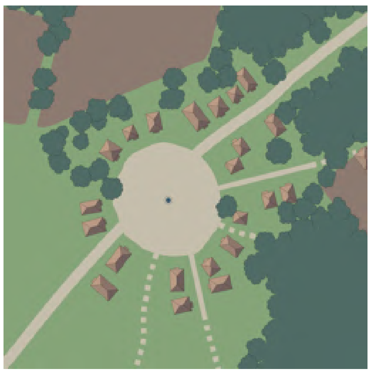
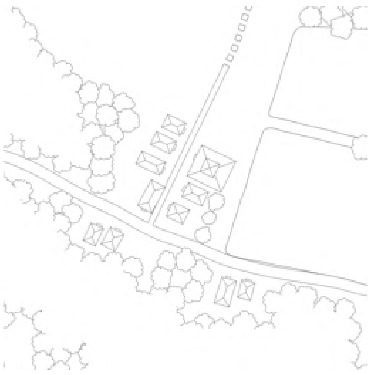
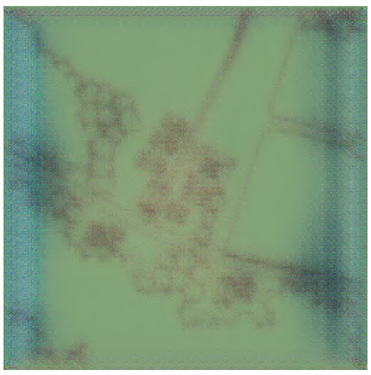
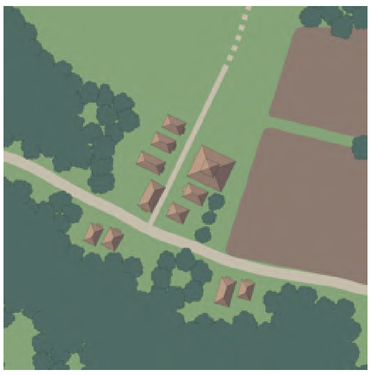
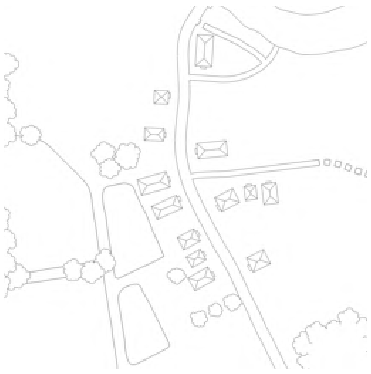

(l) Expected Output

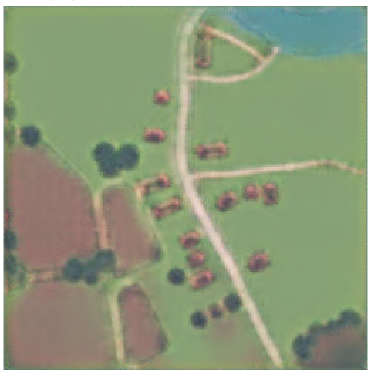
Figure 3.11: Training progress with Batch Size of 5 after each 32 epochs, starting from 4

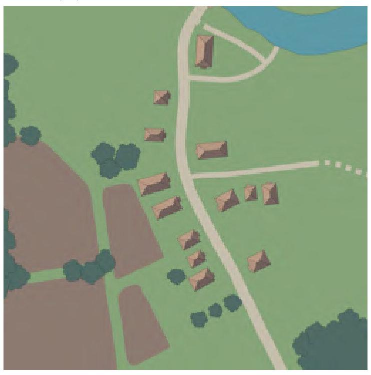

(a) 4-Epochs Input Edges

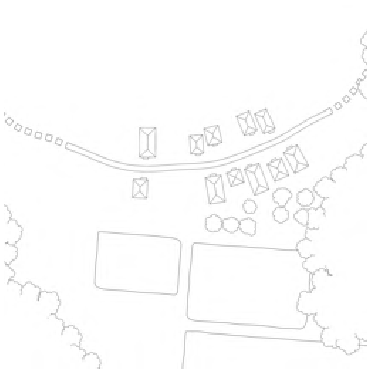

(b) Generated Output

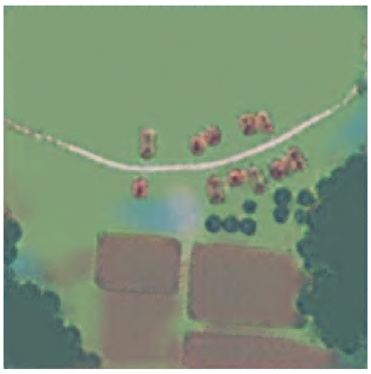

(c) Expected Output

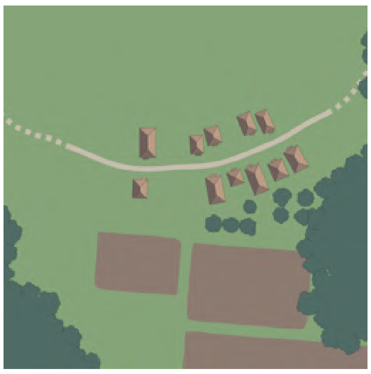

(d) 36-Epochs Input Edges

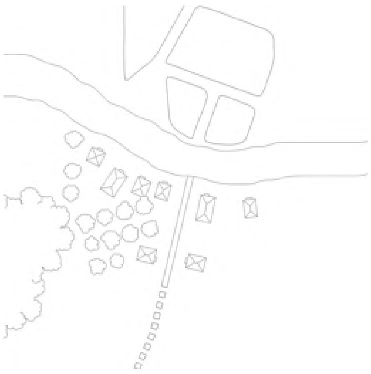

(e) Generated Output

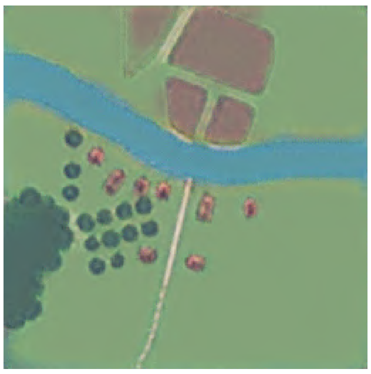

(f) Expected Output


(g) 68-Epochs Input Edges


(h) Generated Output


(i) Expected Output


(j) 100-Epochs Input Edges


(k) Generated Output

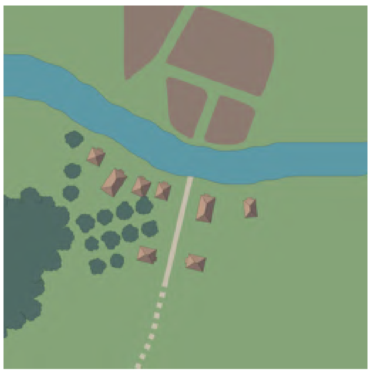
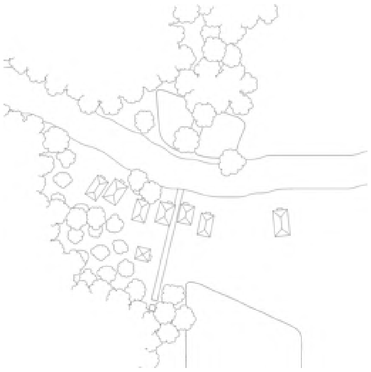
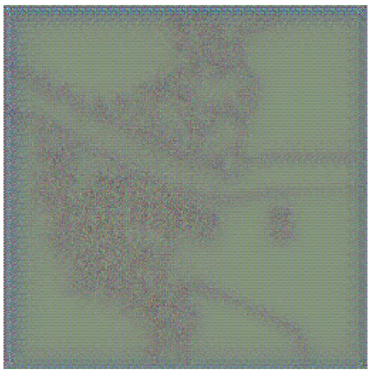
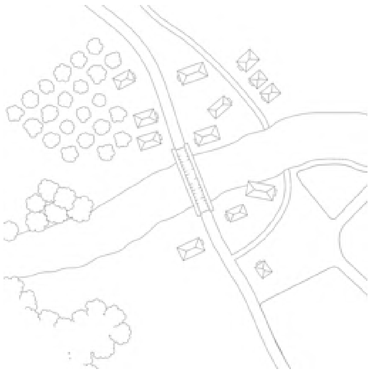

(l) Expected Output

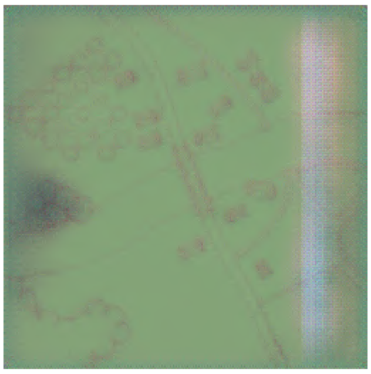
Figure 3.12: Training progress with Batch Size of 10 after each 32 epochs starting from 4


(a) 4-Epochs Input Edges

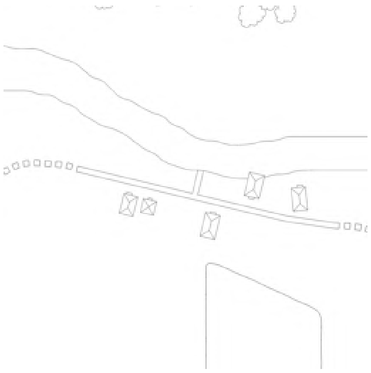

(b) Generated Output

(c) Expected Output


(d) 36-Epochs Input Edges

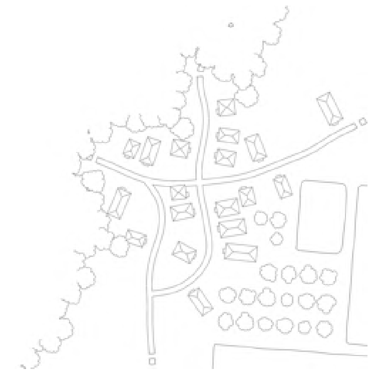

(e) Generated Output

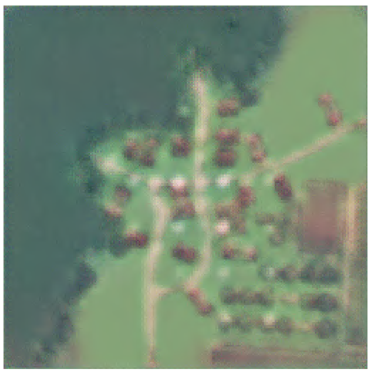
(f) Expected Output


(g) 68-Epochs Input Edges

(h) Generated Output

(i) Expected Output


(j) 100-Epochs Input Edges


(k) Generated Output

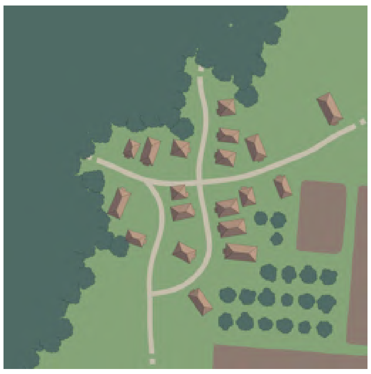
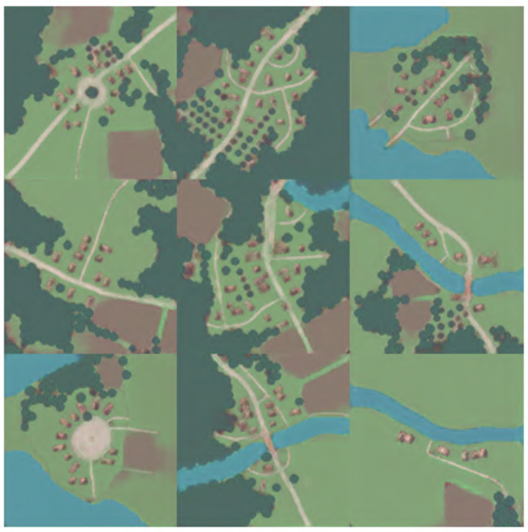
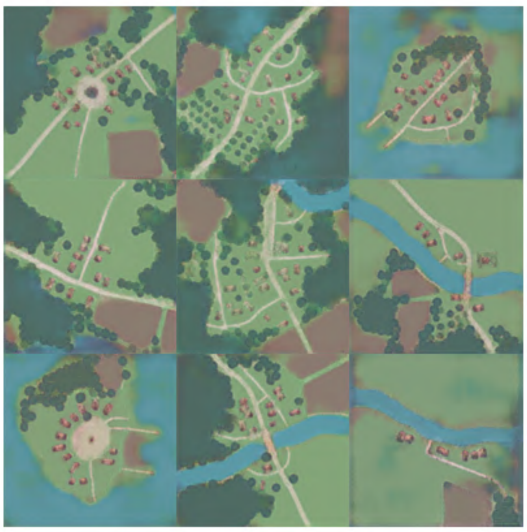
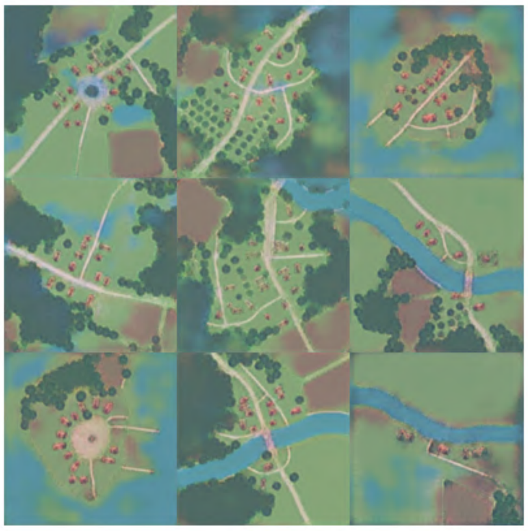

(l) Expected Output

Figure 3.13: Training progress with Batch Size of 50 after each 32 epochs starting from 4


(a) Outputs with Batch Size of 1


(b) Outputs with Batch Size of 5


(c) Outputs with Batch Size of 10

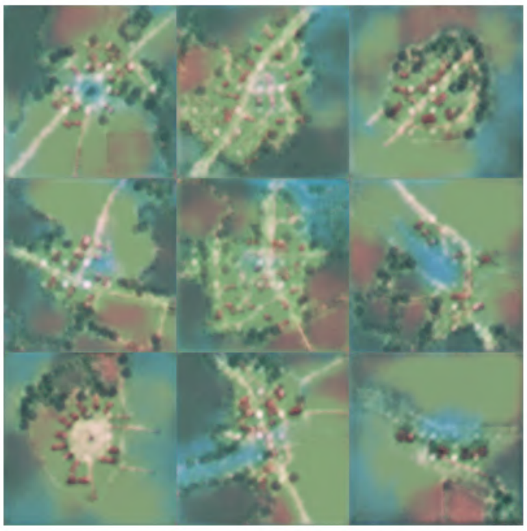

(d) Outputs with Batch Size of 50

Figure 3.14: Comparison of Generated Outputs after 100 Training Epochs with Different Batch Sizes

3.4 Translating from Drawings

It is important to note that the generated outputs displayed in Section 3.3 only demonstrate the model's generative capabilities on previously seen data. As the model is already familiar with this data, the outputs tend to be more aligned to the expected output. In this manner, to truly assess the effectiveness of the model in solving the intended problem, generating images from hand-drawn sketches, it becomes necessary to evaluate its performance on novel data. In the following sections I will introduce the drawing application I developed using Pythons tkinter library, which enables the user to instantly submit hand-drawn sketches to the Pix2Pix model and display the generated map. By the end, I will provide my conclusions and reflections based on output observations. Figure 3.15 illustrates the workflow of the experiment, where, from the drawing application, the sketch of a map is submitted to the Pix2Pix network, trained in section 3.3. The network processes this sketch and returns the generated image to the application.

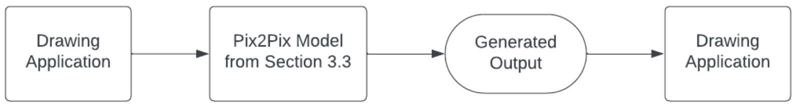


Figure 3.15: Diagram: submitting drawing to the AI

3.4.1 Test Environment

As explained previously, the problem I aim in solving is to transform a map sketch into its realistic representation. Although the current implementation of the network is trained on map edges rather than sketches, I decided to assess its effectiveness when provided with hand-drawn inputs.

To make this process more interactive, I developed a Graphical User Interface [1] (GUI), using Pythons tkinter library, that allows me to draw using a mouse and keyboard and instantly feed the drawing as input to the network. The images in Figure 3.16 demonstrate the environment, both when it is in an empty state and after I made drawings in it. The left portion of the environment displays a canvas where I can draw using the mouse and keyboard, while the right side shows the output from the Pix2Pix generator.

3.4.2 Test Results

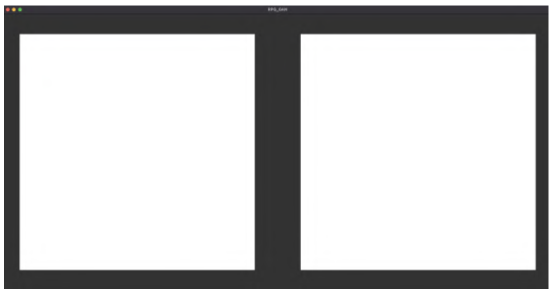
Upon examining the results, it is evident that the model exhibits a high degree of effectiveness in generating trees, as shown in Figures 3.16b and 3.16c.

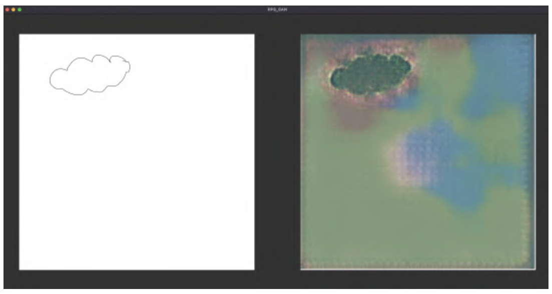
[1] link to demonstration: `https://youtu.be/RyXdK7xePrY`

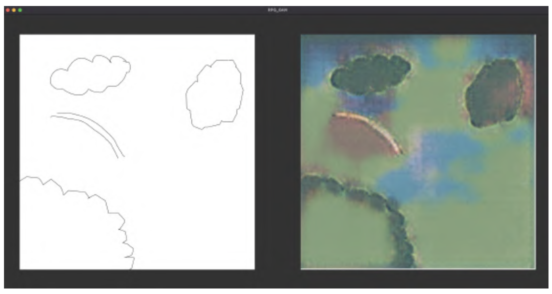
Additionally, Figures 3.16d and 3.16e demonstrate the models' ability to create roads, a town square, houses and patches of land.

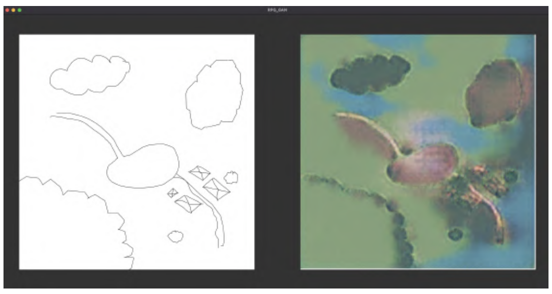
However, upon closer observation of Figure 3.16f, it becomes apparent that there is room for improvement towards generating houses. If the size of the drawn house is too large, the network struggles to comprehend its structure and ends up generating bad patches of land. Similarly, when it was too small, the network mistook it for a squared tree.

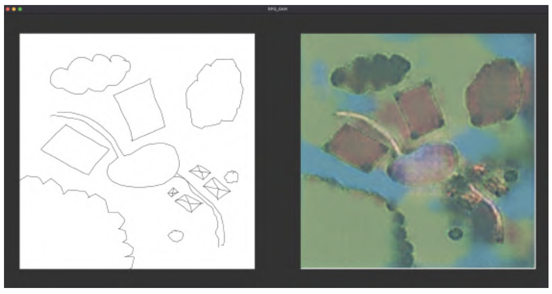
Additionally, Figure 3.17 displays a cherry-picked generated result after numerous attempts, where the model correctly interpreted each drawn element.


(a) clean screen


(b) one tree


(c) trees and road


(d) houses and town square


(e) patches of land

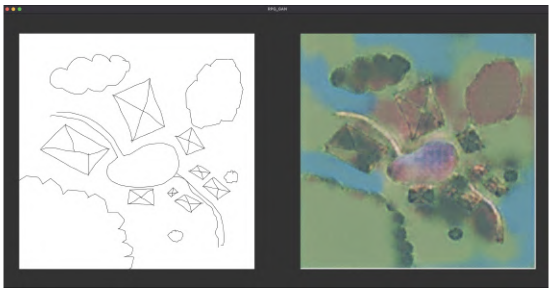
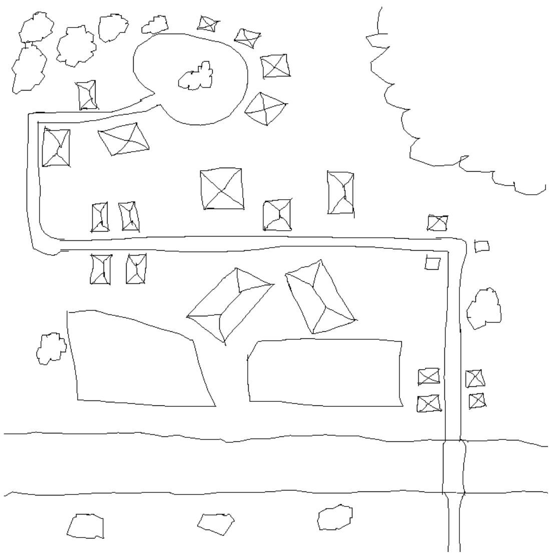
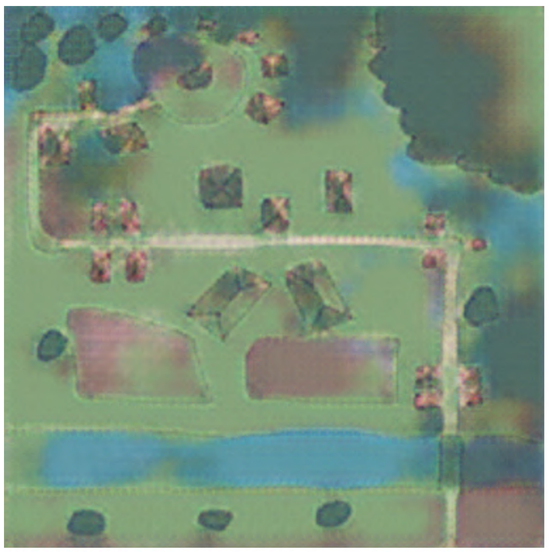

(f) bad houses and land

Figure 3.16: Network Outputs from Hand Drawings


(a) Drawing Input


(b) Generated Output

3.5 Training with Hand-Drawn Sketches

Building upon the success of the previous experiment, I decided to take it even further beyond. While, previous experiments relied on clever techniques to process input data and mimic map hand-drawn sketches, they proved less effective while dealing with images of higher resolution. For this experiment, in order to overcome this issue, I aimed to directly train the model using hand-drawn sketches and evaluate its effectiveness in generating maps from them.

In the following sections, I will explain how I drew and processed these sketches. I will go into the challenges I encountered along the way, the strategies I employed to overcome some of them, and propose solutions for unresolved issues, which could be explored in future iterations. Figure 3.18 presents a general overview of this experiment, where I will manually create sketches of map asset images. These sketches, along with their corresponding original representations, will form the Sketch-Asset dataset, which will be used to train the Pix2Pix network.

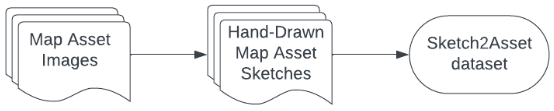


Figure 3.18: Diagram: Hand-Drawing sketches to create Sketch-Asset dataset

3.5.1 Creating Hand-Drawn Assets

To begin, I aimed to generate map assets: the small, more important objects, such as rocks, trees and grass. Figure 3.19, downloaded from the internet, illustrates an example of some of these assets in their original form. Each asset was hand-drawn using a pencil and paper, and subsequently scanned using a printer, resulting in Figure 3.20. In order to remove unnecessary noise from the hand-drawn sketches and make them more similar to what I'd use as input in the testing phase, I thresholded each drawing, converting it into a black and white image, resulting in the rightmost element depicted in Figure 3.21.

Similar to previous training iterations, I concatenated each colorized asset with its corresponding counterpart to create the training dataset. An example taken from the resulting dataset can be seen in Figure 3.21. However, due to the limited number of assets available and manpower to draw these images, the dataset ended up containing only 26 distinct asset-drawing image pairs. Consequently, an entire training phase with 100 epochs didn't allow for the model to gather sufficient feedback over the data and effectively update its parameters. To compensate for this, I trained the model for 600 epochs until I obtained visually satisfactory results, represented in Figure 3.22.

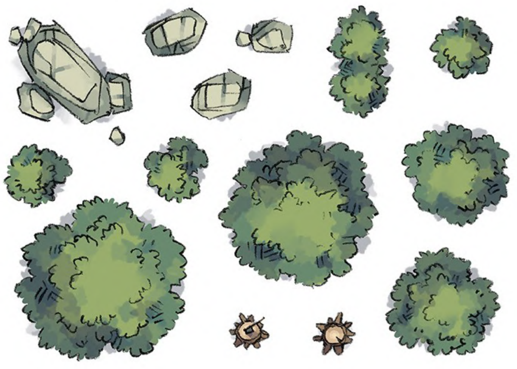


Figure 3.19: Map assets scraped from the internet.

3.5.2 Results

Following the training phase, I proceeded to test the Pix2Pix model using the dedicated test environment tool that I developed and explained in Section 3.4.1. Upon examining the outputs, it became apparent that the small dataset size had a large impact on the results. As demonstrated in the images in Figure 3.23, the model appears to be skewed towards generating rocks. Where we would expected a vibrant green color, we instead observe a mossy blend of gray and green tones.

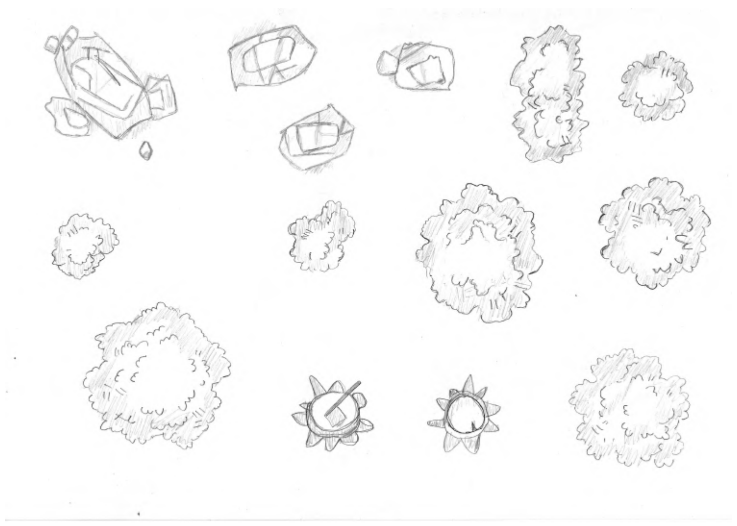


Figure 3.20: Scanned hand-drawn map assets

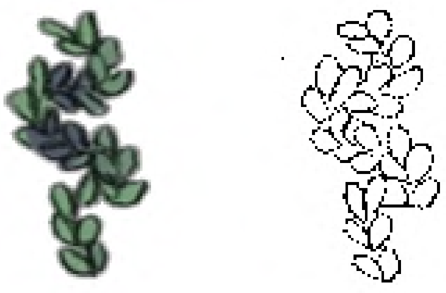
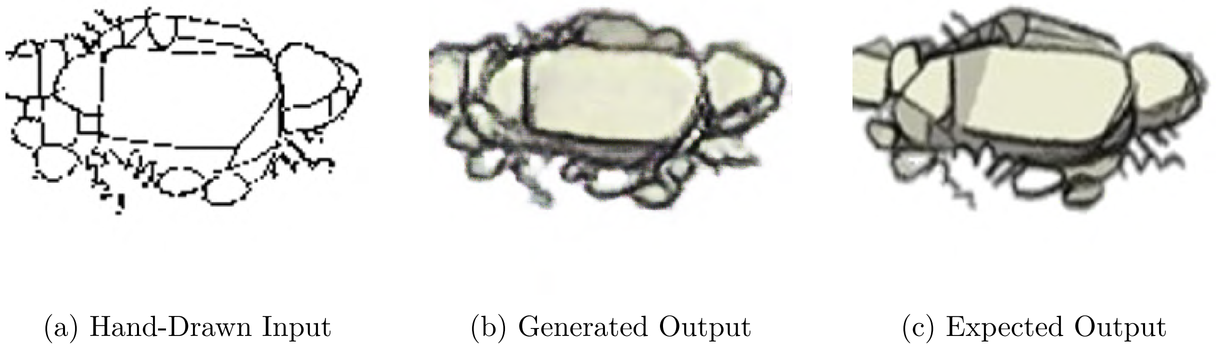


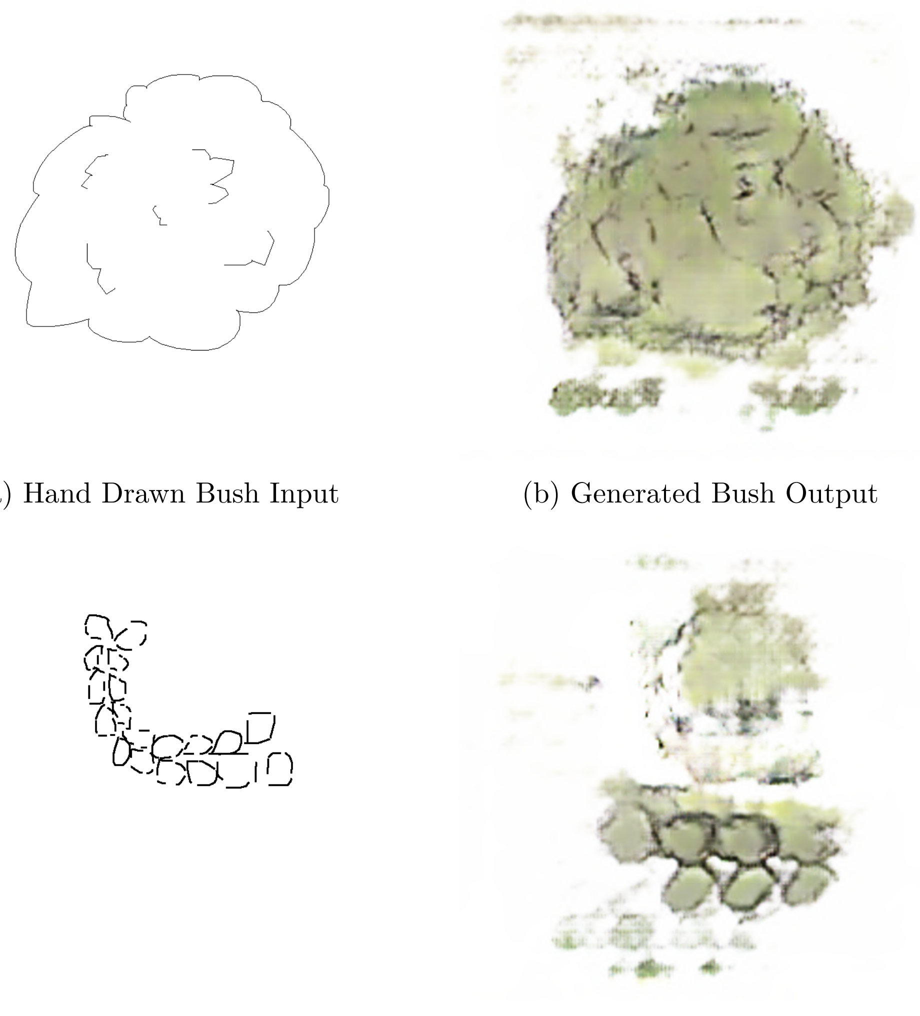
Figure 3.21: Original asset and its thresholded version.



(a) Hand-Drawn Input (b) Generated Output (c) Expected Output

Figure 3.22: Hand-Drawn Input, Generated Output and Expected Output

It is worth considering that the strokes in the sketches differed from both the training and test images, contributing to to the output discrepancy. To address this issue, a more effective approach would involve creating the training dataset directly within the test environment, ensuring better alignment between the training and testing inputs.


(a) Hand Drawn Bush Input (b) Generated Bush Output

(c) Hand Drawn Leaves Input (d) Generated Leaves Output

Figure 3.23: Example Input and Outpus

3.6 Training with Larger Maps

In the previous section I left as an open issue the need for a larger dataset to improve the model's performance. However, due to the lack of resources for manual drawing, I decided to address this issue from another perspective. In this experiment, I explore training the model with maps of higher resolution that contain more information and details. My hope is that the additional information within each image will improve training without the need for a larger dataset. In the following sections, I will dive deeper into detailing how I created this new dataset and by the end share my thoughts on the generated outputs. Figure 3.24 presents a general overview of this experiment, where I will manually create sketches of map images. These sketches, along with their corresponding original representations, will form the Sketch-Map dataset, which will be used to train the Pix2Pix network.

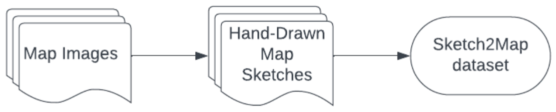


Figure 3.24: Diagram: Hand-Drawing sketches to create Sketch-Map dataset

3.6.1 Creating Hand-Drawn Maps

One limitation of the previous experiment was the lack of images in the training dataset and consequently lack of information for the model to train and improve upon. To overcome this, in this experiment, instead of increasing the dataset length I focused on increasing the amount of information within each map image.

To accomplish this, I searched the internet for detailed RPG maps, such as the one shown in Figure 3.25. These maps contain many more features and details compared to the ones from the previous experiment. In a piece of paper, placed on top of a screen displaying the map, I traced the most important features using a pencil. After that, I added some more details, such as leaves and water currents, resulting in the hand-drawn map shown in Figure 3.26. Once this was done, I scanned every drawings and saved them on disk. Like before, the original and hand-drawn images were concatenated and used to train the model. The final dataset consisted of 16 map-drawing image pairs. Each map image had a significantly higher resolution and more information than the assets used in the previous experiment.

3.6.2 Training

Using the same implementation and test script as the previous experiment, I trained the Pix2Pix model with the new map-drawing dataset. Interestingly, each training epoch lasted slightly longer, approximately 20 seconds, when compared to the previous experiment, despite the smaller dataset and inputs being cropped to have the same size. This difference in duration could be a consequence of the higher degree of details in this experiment's training data, but further investigation is needed to explore the reasons behind this variation. Beyond that, satisfactory results were achieved without the need for extending training for as many epochs as the previous experiment. After 100 epochs of training, the model captured the most prominent features of Figure 3.26. Extending the training for a longer period offered little advantage to the outputs. Figure 3.27 illustrates the model's evolution towards generating images throughout the training process, with snapshots taken at every 50 epochs until epoch 300.

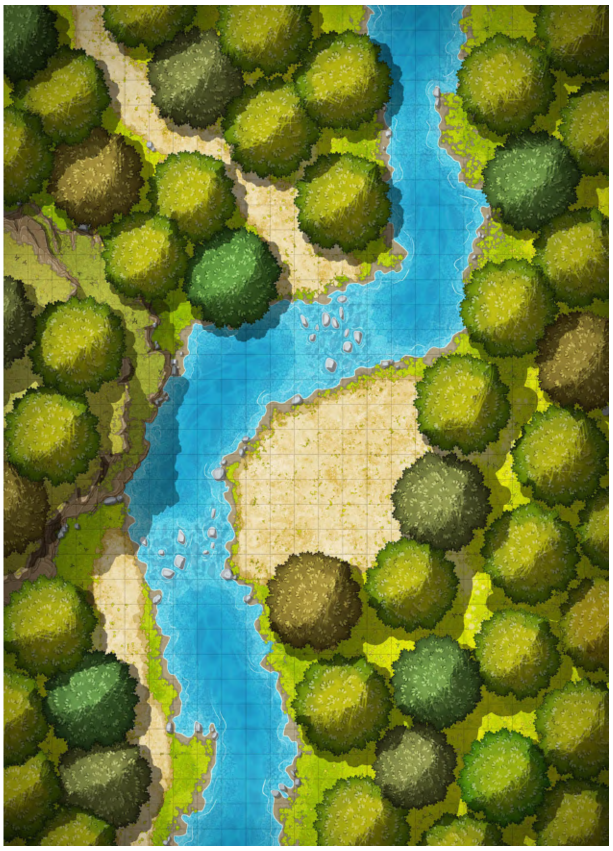


Figure 3.25: Map found online.

3.6.3 Testing

Learning from previous mistakes, I wanted to ensure that the testing process aligned with the training methodology. To achieve this, I had to create a set of brand new maps using the same approach I used when creating the training dataset. I hand-drew a couple of maps and scanned them using a printer. These scanned images would serve as input for testing the network. In order to better evaluate the network's evolution, each drawing was incrementally modified to include additional details, such as grass patches, trees or hills. The scanned drawings, used as input, and corresponding generated output images can be seen in Figure 3.28.

One significant drawback that became apparent during the testing phase was the lack of an instantaneous output feedback from the network, that prevented me from adjusting the drawing in real-time. For example, observing Figure 3.28f, we can see that the network successfully identified and generated the vertical road, but failed to identify the road that runs horizontally across the image, generating a large patch of grass instead. Since both roads only differ in width, if I had received this feedback earlier, I could've redrawn the bad road to be thinner. This limitation delayed the testing process making it much less

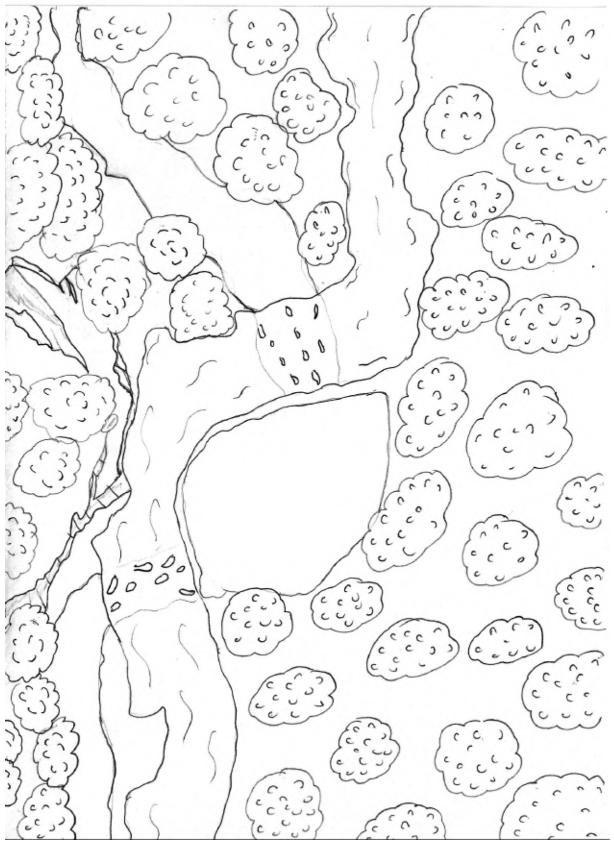
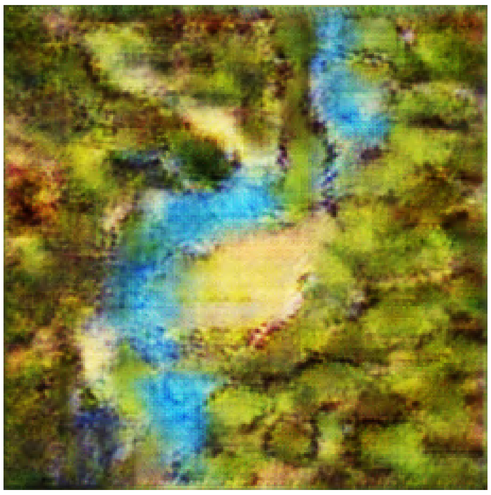


Figure 3.26: Hand Drawn Map.

effective.

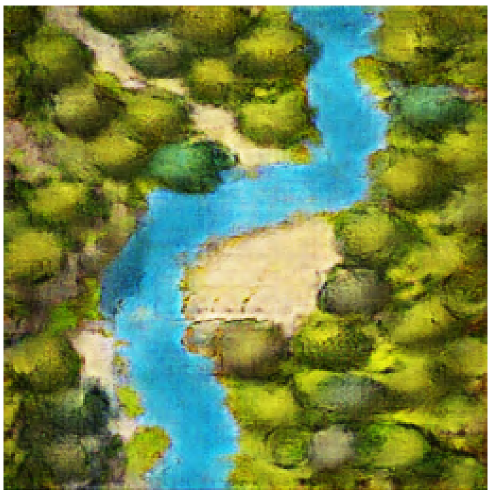
Beyond that, an interesting phenomenon can be observed when comparing the generated outputs. The quality of the output image drastically improves once the entire input drawing contains some level of detail. For example, when we compare Figures 3.28b and 3.28d, where only a couple of trees were additionally drawn at the bottom of the image, both outputs appeared similar. However, when we compare Figures 3.28d and 3.28f, where the entire image contained some degree of detail, the output image exhibited much more detail and vibrant colors. This behavior can likely be explained from the fact that every image used in the training phase contained details spread across the entire image, leading to the network's struggle on how to interpret empty white space.


(a) after 50 epochs

(b) after 100 epochs

(c) after 150 epochs

(d) after 200 epochs


(e) after 250 epochs

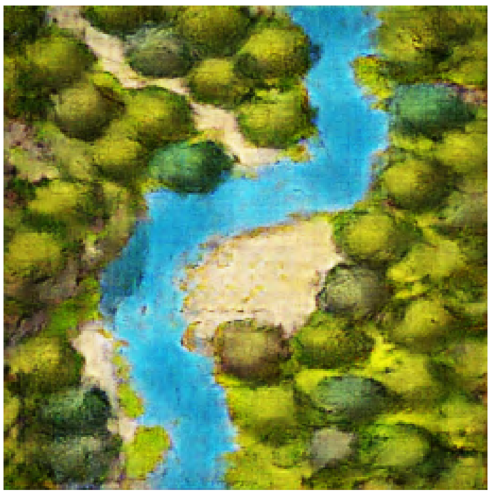
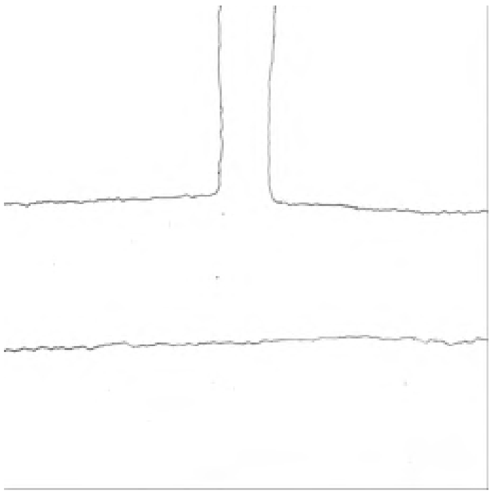
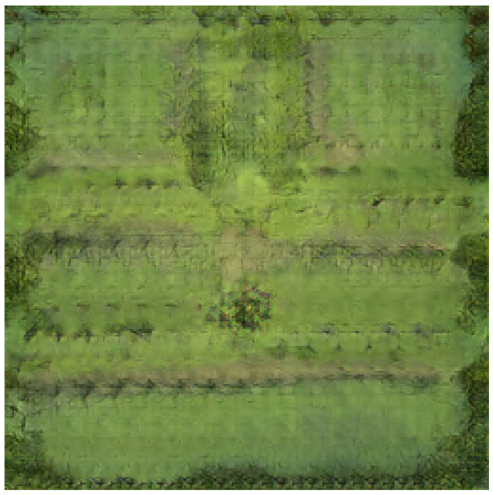

(f) after 300 epochs

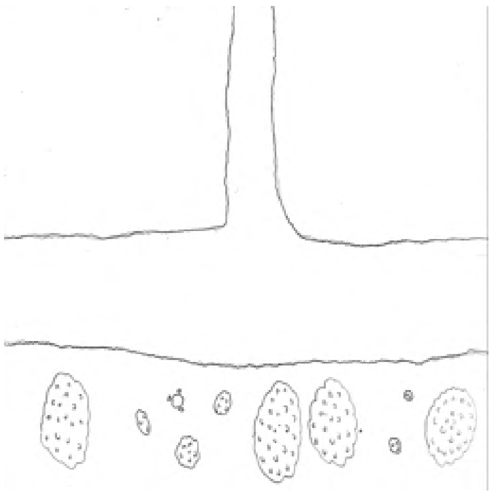
Figure 3.27: Network Outputs from Hand Drawings



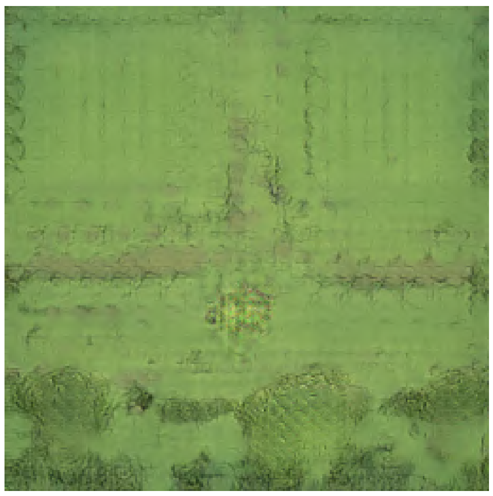
(a) Input Drawing



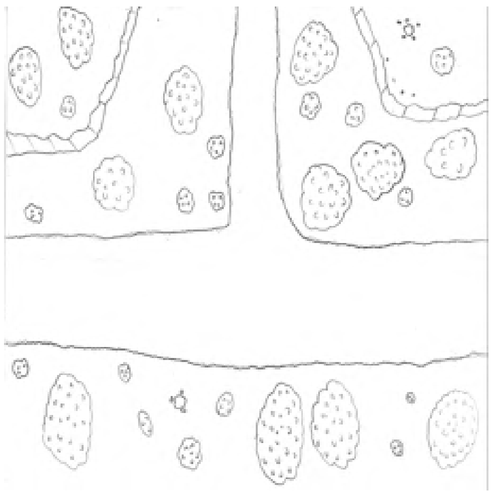
(b) Generated Output



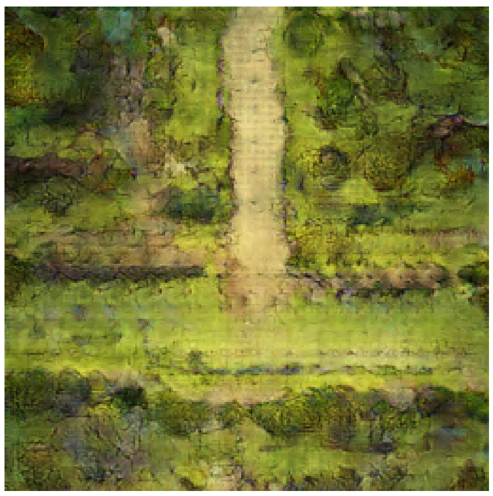
(c) Input Drawing



(d) Generated Output



(e) Input Drawing



(f) Generated Output

Figure 3.28: Network Outputs from Hand Drawings

3.7 Evaluation of the Models

While the previous sections' purpose was displaying outcomes and improvements achieved over each iteration of the experiment, this section focuses on discussing the evaluation of the model.

As explained in 2.14, two common methods for numerically evaluating GANs are the Inception Score (IS) and Frechet Inception Distance (FID). Both methods rely on using a separate neural network as an auxiliary tool to compute an overall score of the GANs output. This auxiliary network is previously trained to detect different kinds of objects within an image, and both IS and FID leverage this network's ability to discern objects to output the quality of a generated fake image.

However, evaluating the Pix2Pix model using IS and FID proved challenging, specifically due to the context of fantasy map generation. The auxiliary network needed to be trained to detect maps and map-related objects, such as trees, roads and buildings. This would have required me to label multiple assets that could appear in the generated maps, which would be a time consuming task.

Because of this, the evaluation of the Pix2Pix model's outputs was conducted through my own visual inspection. This subjective evaluation was the main driving force behind each improvement I made in the experiment. My own accession of quality led me to train the model longer as well as change the inputs.

Chapter 4

Conclusions and Closing Remarks

Throughout this work I have explored the application of Generative Adversarial Networks in image generation, specifically in the context of translating hand-drawn map sketches into fantasy maps using the Pix2Pix model. Initially, I provided the theory required to understand how the model worked and, by the end, I got into the development process detailing how the model was trained, the problems I encountered, how I managed to work around them and conclusions drawn from each iteration of the experiment.

While my initial expectation was to create highly detailed fantasy maps, the overall results were quite satisfactory. The initial experiment served as a quick way to test the model's efficiency before moving on to the problem at hand. The results obtained from this initial experience were surprisingly impressive, despite using a training dataset that differed significantly from the proposed test input. The model demonstrated the ability to accurately interpret and generate land, water, roads and even features that resembled houses. It was impressive to achieve such detailed output with relatively little effort. The success of the initial experiment played a crucial role in motivating further exploration and refinement of the model.

Following that, I refined the model by employing a training dataset specifically tailored for the problem at hand. Once again, I was impressed by the results. In Python, I coded a simple script that efficiently extracted the borders from images, resulting in a large dataset. Training was fast, and detailed images were quickly generated by the network. To test the model, I developed a training environment that not only enhanced the effectiveness of testing but also provided a fun experience for the user. However, in hindsight, using a testing environment different from the one used during training and the human error to draw straight lines had a significant impact on the model's ability to interpret the edges accurately, resulting in reduced image quality. Overall, I believe that the test environment played a crucial role in preserving a sense of personal involvement and ownership over the final output, despite the assistance of Artificial Intelligence. It allowed users to maintain a connection with the creative process while benefiting from the augmentation provided by the model.

In the final experiments, where I tried to increase the degree of detail by hand drawing high resolution assets and maps, although ambitious for such a small time, provided valuable insights of what the model can be trained to achieve. These limitations particularly regard dataset size and differences of platform used in training and during testing. Given more time and a larger development team, it would've been possible to scrape and draw more images for the training phase. Additionally, with the help from a team of designers, the maps and their corresponding sketches could've been directly created in the drawing application, resulting in consistent assets across maps, which could've facilitated the learning process for the model. Also, had I known about the difference of performance caused by using different platforms in the testing phase, I could've integrated the entire process to the same platform, which would have probably given better translations.

Despite the impact Pix2Pix had in the world of Generative networks, it is worth noting that newer models have emerged that promise output with greater quality. One such model is Pix2PixHD, released by *NVIDIA*, which synthesizes high-resolution images with up to 2048x1024 pixels. Training the model, however, required significant time and GPU power, which would've significantly delayed production of this work.

Another model, CycleGan, which was also introduced by the Pix2Pix team, offers the possibility to solve translation tasks using an unpaired image dataset. Unlike Pix2Pix, which requires paired images that represent the same image but in different domains, as input during training, CycleGAN proposes a novel training process that does not require paired representations. While this other model presented intriguing possibilities, training it would've also been time consuming, prompting my decision to move away from it and focus on Pix2Pix.

Future work in this area could explore the use of these newer models and compare their quality and performance from the baseline results achieved with Pix2Pix. Additionally, the performance of the currently used Pix2Pix network can be further improved by creating a better training dataset, based in insight gained with the project.

In terms of evaluation methods, while my own subjective evaluation played a significant role, future research can explore incorporating quantitative metrics, such as FID and IS, as well as user opinions, to provide better analysis of the model's performance.

In conclusion, I believe that my goal was partially achieved. While the generated maps didn't meet the same level of detail as those available on the internet, they reached a point in which they outperform a plain hand-drawing and gameplay is possible. With this in mind, an interesting next step would involve making this tool accessible online and inviting players to test it on their own campaigns.

Referências

[1] OpenAI: *Dall.e-2*. https://openai.com/dall-e-2, visited on 2023-06-10. x, 1

[2] Isola, Phillip: *pix2pix*. https://github.com/phillipi/pix2pix, visited on 2023-06-10. x, 2, 33, 39

[3] Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio: *Generative adversarial nets*. NIPS'2014, 2014. x, 8, 10, 13, 14

[4] Mirza, Mehdi and Simon Osindero: *Conditional generative adversarial nets*, 2014. x, 14, 15

[5] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox: *U-net: Convolutional networks for biomedical image segmentation*, 2015. x, 22, 23

[6] ResearchGate: *Single-shot fringe projection profilometry based on deep learning and computer graphics*, 2021. https://www.researchgate.net/publication/349487608_Single-shot_fringe_projection_profilometry_based_on_Deep_Learning_and_Computer_Graphics, visited on 2023-07-10. x, 25

[7] Watabou: *Village generator*. https://watabou.itch.io/village-generator, visited on 2023-06-10. xi, 34, 37, 38

[8] OpenAI: *Dall · e: Creating images from text*. https://openai.com/research/dall-e, visited on 2023-06-16. 2

[9] Borji, Ali: *Generated faces in the wild: Quantitative comparison of stable diffusion, midjourney and dall-e 2*, 2023. 2

[10] Wikipedia, the free encyclopedia: *Dungeons & dragons*. https://en.wikipedia.org/wiki/Dungeons_%26_Dragons, visited on 2023-06-15. 2

[11] Gui, Jie, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye: *A review on generative adversarial networks: Algorithms, theory, and applications*. CoRR, abs/2001.06937, 2020. https://arxiv.org/abs/2001.06937. 6

[12] Ng, Andrew Y. and Michael I. Jordan: *On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes*. NIPS'2001, 2001. 6, 7

[13] Brownlee, Jason: *Neural networks are function approximation algorithms*. https://machinelearningmastery.com/neural-networks-are-function-approximators/, visited on 2023-06-16. 9

[14] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville: *Deep Learning.* MIT Press, 1st edition, 2016. 12

[15] Arjovsky, Martin and Léon Bottou: *Towards principled methods for training generative adversarial networks*, 2017. 16

[16] Radford, Alec, Luke Metz, and Soumith Chintala: *Unsupervised representation learning with deep convolutional generative adversarial networks*, 2016. 16, 17

[17] Arjovsky, Martin, Soumith Chintala, and Léon Bottou: *Wasserstein gan*, 2017. 17, 19

[18] Thickstun, John: *Kantorovich-rubinstein duality.* `https://courses.cs.washington.edu/courses/cse599i/20au/resources/L12_duality.pdf`, visited on 2023-06-10. 18

[19] Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville: *Improved training of wasserstein gans*, 2017. 19

[20] Miyato, Takeru, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida: *Spectral normalization for generative adversarial networks*, 2018. 20

[21] Pang, Yingxue, Jianxin Lin, Tao Qin, and Zhibo Chen: *Image-to-image translation: Methods and applications*, 2021. 20

[22] Isola, Phillip, Jun Yan Zhu, Tinghui Zhou, and Alexei A. Efros: *Image-to-image translation with conditional adversarial networks*, 2018. 22, 23, 39

[23] Salimans, Tim, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen: *Improved techniques for training gans*, 2016. 26

[24] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, and Jonathon Shlens: *Rethinking the inception architecture for computer vision*, 2015. 26

[25] Heusel, Martin, Hubert Ramsauer, Thomas Unterthiner, and Bernhard Nessler: *Gans trained by a two time-scale update rule converge to a local nash equilibrium*, 2018. 27

[26] Gonzalez, Rafael C. and Richar E. Woods: *Digital Image Processing.* Pearson, 4th edition, 2018. 27, 29

Appendix A

The Python implementation code for both Pix2Pix and the Drawing Application as well as the created datasets are available on GitHub. You can access the repositories at `https://github.com/Tubar2/pix2pix` and `https://github.com/Tubar2/pix2pix-gui`. Feel free to explore the code!