Spring 2024

Bringing GANs to Medieval Times: Manuscript Translation Models

Tonilynn M. Holtz

BRINGING GANS TO MEDIEVAL TIMES: MANUSCRIPT TRANSLATION MODELS

by

TONILYNN HOLTZ

(Under the Direction of Ionut E. Iacob)

ABSTRACT

The Generative Adversarial Networks (GAN) recently emerged as a powerful framework for producing new knowledge from existing knowledge. These models aim to learn patterns from input data then use that knowledge to generate output data samples that plausibly appear to belong to the same set as the input data. Medieval manuscripts study has been an important research area in the humanities field for many decades. These rare manuscripts are often times inaccessible to the general public, including students in scholars, and it is of a great interest to provide digital support (including, but not limited to translation and search) for accessing these materials. We propose a GAN framework that uses manuscript images and their translations to create a model capable of new translation from new manuscript images. Such a model would provide great assistance to humanities researchers seeking to produce digital editions of old manuscripts.

INDEX WORDS: Generative Adversarial Networks, image generation models, medieval manuscripts, machine learning, digital editions

2009 Mathematics Subject Classification: 68T07, 68T10, 62H35

BRINGING GANS TO MEDIEVAL TIMES: MANUSCRIPT TRANSLATION MODELS

by

TONILYNN HOLTZ

B.S., Georgia Southern University, 2022

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

BRINGING GANS TO MEDIEVAL TIMES: MANUSCRIPT TRANSLATION MODELS

by

TONILYNN HOLTZ

Major Professor:	Ionut E. Iacob
Committee:	Yan Wu
	Scott Kersey

Electronic Version Approved:
May 2024

DEDICATION

To my parents and brother, whose love and encouragement sustained me through my educational path, offering guidance and wisdom whenever I needed it.

To my friends who always cheered me on, reminding me I've got what it takes and to keep my eyes on the finish line.

And lastly, to my devoted fiancé, whose unwavering support and strength fueled my journey. I owe much of my success to his boundless encouragement and fortitude.

ACKNOWLEDGMENTS

I wish to acknowledge my committee chair, Dr. Emil Iacob, for his patience and guidance. Your advice has been invaluable and I will be forever grateful for the support you provided every step of the way.

I would also like to acknowledge my committee members, Dr. Yan Wu and Dr. Scott Kersey, for their willingness to participate in my defense and for taking the time to review my thesis. Not only have you been outstanding professors, but you've also served as inspiring mentors.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF SYMBOLS

$\mathbb{R}$	Real Numbers
$\mathbb{R}^{H \times W \times C}$	Real valued 3D array
$\mathbb{Z}$	Integers
$\mathscr{I}, \mathscr{T}$	3D arrays representing images
$\mathscr{M}$	The parametric image translation model
$\mathscr{DS}$	The source images dataset
$\mathscr{DT}$	The target images dataset

CHAPTER 1

INTRODUCTION

The study of medieval manuscripts is an area of great interest in the field of humanities. These historical pieces give a unique insight into the culture, lifestyles, and minds of the people whose lives are so distant, that they almost seem fictional. However, many limitations make it difficult for the general population to read and perform studies of their own. Over the years technological advances have, and continue to, fix and address some of these shortcomings. Providing digital support for these manuscripts is extremely important as it provides new ways to utilize them, better legibility, a secured form of preservation, and easy access for a much larger population. Digital support can come in many forms including, but not limited to, translation, digitization, and making the transcripts searchable.

One of the most popular manuscripts is the epic poem Beowulf. Historians believe that the poem was originally written between 975 CE and 1025 CE in England. The manuscript is written in Old English and there is only one known surviving copy. In the 17th century, the Cotton Library, a room housing many medieval manuscripts including Beowulf, erupted into flames. While the Beowulf manuscript made it out of the fire with minimal damage, the same can not be said for many of the other manuscripts being held there. Despite rigorous maintenance and preservation attempts, the original manuscript is fragile and has retained a good bit of damage. Creating a digital form of these important manuscripts provides a more permanent copy addressing one of the major issues humanities researchers face.

Generative Adversarial Networks were first proposed by Ian Goodfellow in 2014. Inspired by a zero-sum mini-max game, Goodfellow's proposed model consisting of a generator and a discriminator trained in competition with one another [7]. The generator is a generative network designed to create forged data of the original data set. Its adversary, the discriminator is oriented to distinguish between the forgeries produced by the generator and the original data. These inner networks are comprised of a series of fully connected layers

or convolutional neural networks. Convolutional GANs are most commonly used in image processing since convolutional neural networks can retain more important information, compared to artificial neural networks, such as spatial dependencies.

For this research, we created and improved a GAN framework that applies image-to-image translation to images of the original Beowulf manuscript. Manuscript images and their digital translations are inputted into the discriminator to train it to distinguish between generated images and real images. Simultaneously the generator is fed random noise and trains to generate translated images of the original manuscript without ever seeing the real transcript images itself.

The goal of this project is to create a model that can intake new manuscript images and produce the corresponding translations.

1.1 Machine Learning

Machine learning is a form of artificial intelligence and a branch of computer science that aims to mimic the way humans learn. Similar to humans, computers learn and improve through experience [3]. The more a model is trained, the more experience it gains, and the better it gets. In the field of study, there are many different machine learning algorithms. These algorithms are trained to find relationships and similarities in the input data and apply the learned information to new data.

Definition 1. Machine Learning is a branch of artificial intelligence that uses algorithms to create models capable of learning to perform tasks without being given explicit instructions.

Machine learning models are not explicitly programmed to carry out a goal. They are designed in a way that they are able to achieve the desired outcomes all on their own. These algorithms can be used to make predictions, classify data, cluster data, and even generate entirely new data. With the rising growth of interest in big data, machine learning has

become highly desirable as it is extremely proficient in dealing with large data sets [3]. At its core, machine learning algorithms are mathematical optimization problems, where the goal is to maximize the accuracy of predictions or classification of new data.

1.2 Supervised Learning

Supervised learning is a type of machine learning that takes labeled data as its input. Labeled data is data that includes both the input and the desired output. The information provided to the model is known as the training data and is represented as a matrix. This matrix is commonly referred to as the feature matrix. There are two main types of supervised learning, classification and regression. Classification learning is applied when the training data is finite, such as color or names. On the other hand, regression is used when the input data is infinite or inside a range.

1.3 Unsupervised Learning

Unlike supervised learning, unsupervised learning uses unlabeled data as its input. Unlabeled data only contains features and not desired outcomes or names. Nowadays, most neural networks and deep learning models use unsupervised learning over supervised. The model is trained to distinguish patterns and relationships among the inputs without any human assistance. This learned information can then be applied to new data to decide whether or not these features are present. The main application of unsupervised learning is clustering.

CHAPTER 2

NEURAL NETWORK MODELS

2.1 Artificial Neural Networks

Neural networks are an important form of machine learning that revolutionized the technological world. The inspiration for artificial networks came from the way neurons in the brain process and transport information. A biological neural network is comprised of billions of neurons. These neurons connect with other neurons to process and transport important signals. Figure 2.1 represents one of the neurons in the brain. The Dendrites take inputted information/signals from other neurons or cells in the body. The information is processed inside the body of the cell [18]. This information is then sent through the Axon to the Axon terminals which connect to another neuron to send output information.

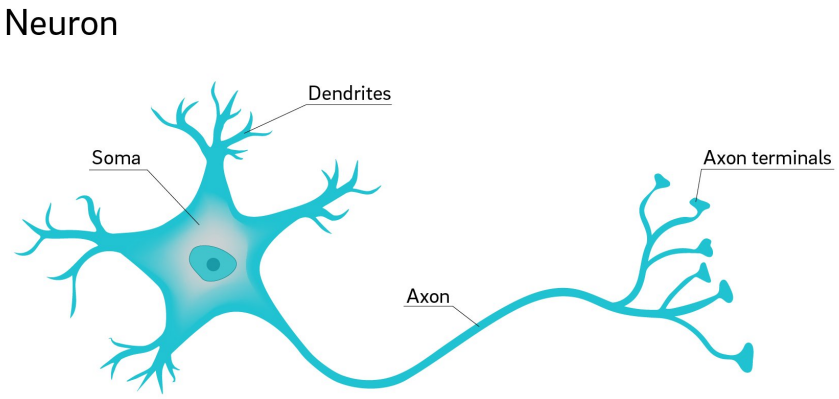


Figure 2.1: Biological neuron

As seen in Figure 2.2, a neuron of an artificial neural network, often referred to as a node, greatly resembles that of the natural neuron. The given inputs x_1, x_2, x_3 are information sent from one neuron to the next. Then inside the neuron, an activation function is

applied to the dot products of the weights and the input values. The weights, represented by w_1, w_2, w_3 are parameters learned through training the network and are calculated through backpropagation. Using this example, let f represent the function mentioned above. Then f is calculated as follows,

$$f(x) = w_1x_1 + w_2x_2 + w_3x_3 + b \tag{2.1}$$

where b represents a bias variable. Each layer in a neural network is given a bias to account for unseen outcomes or patterns.

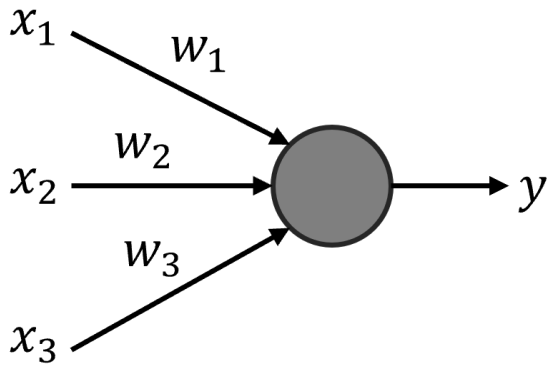


Figure 2.2: Artificial neuron

Similar to a human brain, a collection of these nodes makes up an artificial neural network. A data vector is inputted into the model as a series of nodes. Every node in the input layer sends output to every node in the following hidden layers. The hidden layers are the bulk of a neural network. Inside these layers are where the information from the previous layer is processed and learned. The learned information is commonly referred to as weights. In the ANN architecture, following the final hidden layer is the output layer. This model can be seen in Figure 2.3.

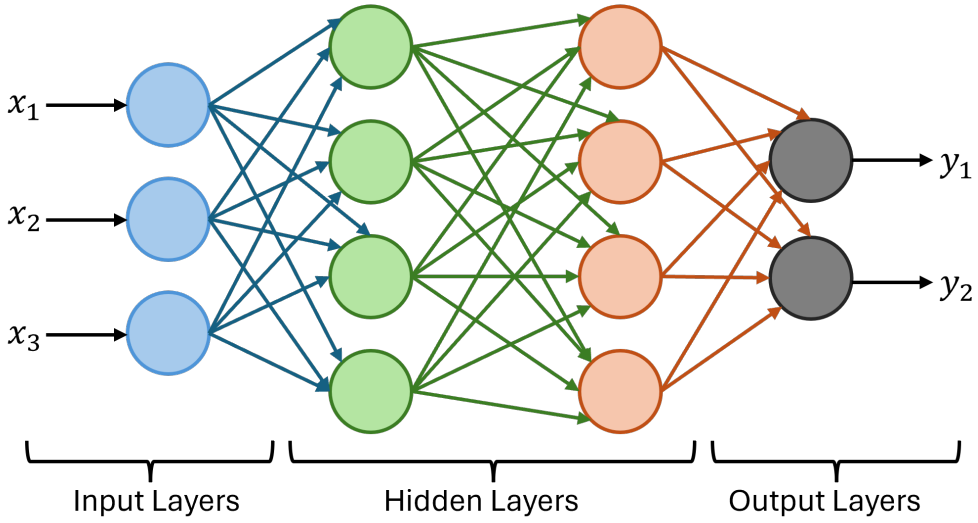


Figure 2.3: A visual representation of an ANN architecture.

2.1.1 Weights

Weights can be described as the relationship between the input and output of each node, where the larger the weight value, the stronger the relationship. At the beginning of the model, weights are initialized as random values. In each node, weights are summed and processed through an activation function [18]. Weights are updated after each epoch of the network using backpropagation. Backpropagation is a form of the gradient descent method that uses the parameters of the network as their reference. The overall goal of an artificial neural network is to optimize the weights to minimize the error/loss function [2].

Definition 2. Backpropagation is a type of gradient descent method used to calculate and update the weights of an artificial neural network's outputs.

2.1.2 Hidden Layers

Although hidden layers are not required for a model to be classified as a neural network, hidden layers are the core of most neural networks. An ANN with only one layer is com-

monly referred to as a ”shallow” network, while ANNs with many hidden layers are classified as deep learning models. These layers are made up of a collection of nodes where the output of one layer represents the input of the following. An ANN can contain any number of hidden layers. This number is entirely dependent on the goal/purpose and complexity of the network [9].

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are deep learning models primarily used in processing images. CNNs take their input and are trained to be able to determine the importance of details in the input to differentiate between different observations. Unlike a traditional artificial neural network, that can only take numeric data, CNNs can be applied to big data such as visual, audio, and textual data. One of the most common applications of CNNs is image recognition. Since an image is a matrix of pixels, with each pixel containing a value that determines its intensity, it would be easy to convert the image into a vector to feed into an artificial neural network. However, this process results in the loss of important information, such as how the pixels around one another interact. By using a series of kernels, CNNs can maintain valuable spatial dependencies [12]. The main purpose of CNNs is to transform an image into different forms that are easier to process without compromising important features and characteristics.

Definition 3. Convolution is a mathematical operation on two functions that produces a third function.

Convolutional Neural Networks are made up of three types of layers: convolutional layers, pooling layers, and fully connected layers.

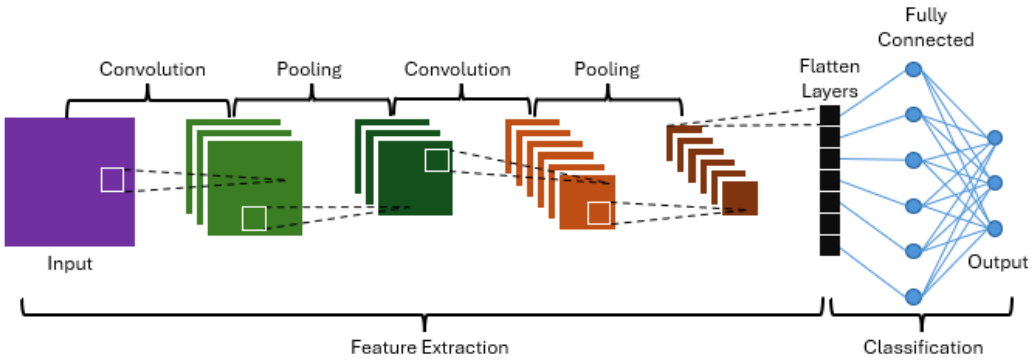


Figure 2.4: A visual representation of a CNN architecture with two convolutional layers.

2.2.1 Convolutional Layers

Convolutional layers are the key component of CNNs. The main focus of this layer is the learnable kernels/filters. A common mistake sometimes made is using kernels and filters interchangeably. Although quite similar there is one key thing setting the two apart. Kernels are two-dimensional matrices whose elements consist of weights that are updated during the training process typically through backpropagation and gradient descent, while a filter is a 3-dimensional collection of kernels [9]. The kernel is applied to the top left portion of an image and the scalar multiplication between the input pixels and the weights in the kernel are calculated [12]. Then the kernel moves to the right by a specified number of pixels, referred to as the stride value until it has reached the end of the image. The kernel then goes back to the beginning of the image, down one stride length, and continues this process until it has made its way through the entire image. The output of this process is known as the activation map.

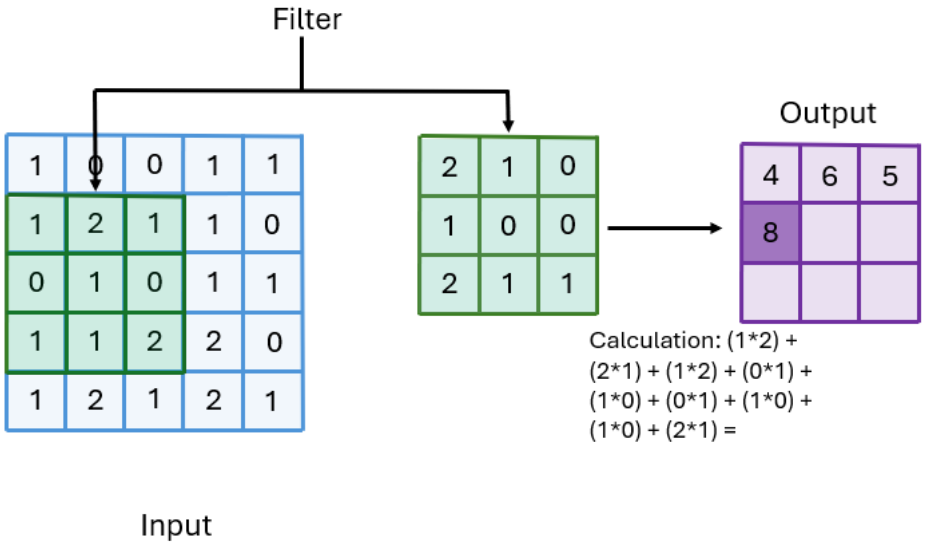


Figure 2.5: Example of how the kernel is applied and the calculation of the scalar multiplication between the input pixel and the kernel.

Every kernel in the layer has an activation map. These maps are then stacked on top of each other to form the output of the entire layer. The size of the output is dependent on three parameters, the number of filters, the stride size, and zero-padding. The convolutional operation sometimes produces a result larger or smaller than the input. This is because the kernels do not always perfectly align with the dimensions of the input. When this occurs, zero-padding is applied. Zero-padding is the process of adding padding to the border of the input [12]. By doing this, the output produced will be greater or equal in size compared to the input.

2.2.2 Pooling Layer

As seen in Figure 2.4, following the convolutional layer is the pooling layer. The pooling layer allows for a significant reduction in the dimensionality of the activation maps [12]. The pooling process takes a window, typically with dimensions 2 x 2, and slides across the

activation maps reducing the number of parameters in each region. There are two types of pooling methods: average pooling and max pooling. In average pooling, the process takes the average of all of the parameters inside the input to produce the output. On the other hand, max pooling only takes the value of the maximum parameter in the region. Max pooling is the most commonly used pooling method in CNNs and is represented in Figure 2.6.

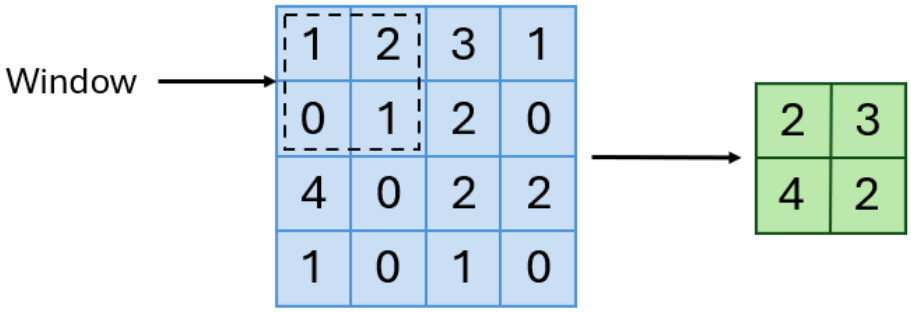


Figure 2.6: Example of max pooling with a 2 x 2 window.

The combination of the convolutional layer and the pooling layer makes up a complete layer in the convolutional neural network. Multiple layers can be used, and when this occurs, the CNN architecture becomes hierarchical [10]. The first layers pull low-level features of an image, such as colors, edges, and angles. These features are typically found on the pixel level. As the model continues into the following layers, the information captured becomes high-level features, such as objects and shapes. The inclusion of high-level features gives a better understanding of the images as a whole. After each convolutional layer, an activation function is applied to the activation maps to introduce non-linearity.

2.2.3 Rectified Linear Unit

A rectified linear unit (ReLu) is a piecewise linear activation function [9]. The function takes a value, x, and returns a positive value or zero. If x is a negative number, the function

returns 0, otherwise the output is the value of x. Let x be the input value, then the formula is shown in equation 2.2.

$$f(x) = max(0, x) \tag{2.2}$$

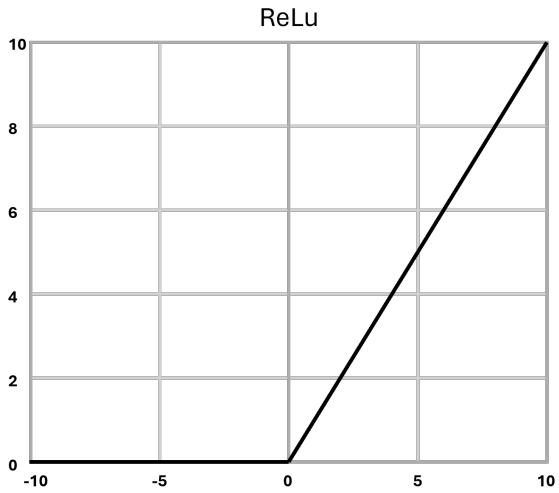


Figure 2.7: Graphical representation of the ReLu function.

Often in CNNs, a leaky ReLu is applied instead. Although similar to a normal ReLu, when a negative number is input into a leaky ReLu function it returns αx in place of zero. This allows better maintenance of the flow of information between the layers. The leaky Relu function is represented in equation 2.3

$$f(x) = max(\alpha x, x) \tag{2.3}$$

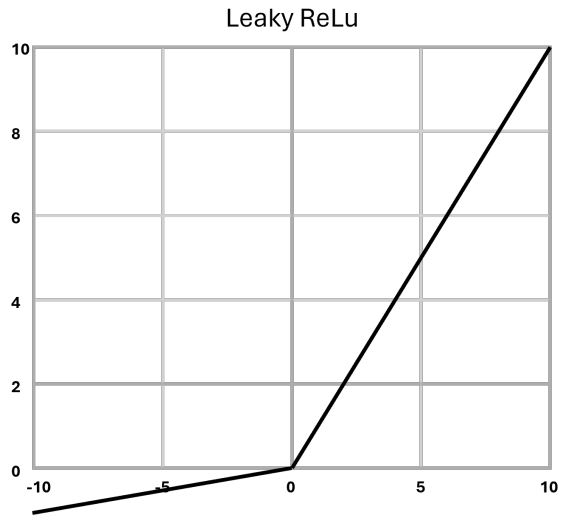


Figure 2.8: Graphical representation of the Leaky ReLu function.

2.2.4 Fully Connected Layers

After the last pooling layer in the CNN architecture are the fully connected layers. These layers are used to flatten the results of the final pooling layer in the network into a vector. Then the vector is processed. The fully connected layers are a way to connect the layers to the surrounding layers and are similar in structure to artificial neural networks [12]. These layers are where tasks such as classification take place and are the final layers in a CNN.

2.3 Generative Adversarial Networks

Zero-sum games are characterized in game theory as a two-person opposition where the gain of one of the players results in the equivalent loss of the other. The core concept of a generative adversarial network (GANs) was inspired by these games where the goal is to find a Nash equilibrium [6]. GANs are a type of system that uses deep learning methods to generate material similar to the inputted data using patterns or commonalities found within. GANs are comprised of two neural networks, the generator and the discriminator.

Both networks are made up of a series of fully connected layers or convolutional layers. Convolutional GANs are more desirable than fully connected GANs when the input data is a series of images [5]. Because of this, we will focus on the composition of convolutional GANs for the remainder of this section.

The generator is an unsupervised model trained to generate new data. Noise is inputted into the generator and it outputs a generated image without ever seeing the desired output images. The discriminator is a supervised model trained to distinguish between the real examples, from the domain, and fake examples, generated. It is fed real and fake images and categorizes them into a number between (0,1). The two models are trained in direct competition with each other until the discriminator can only differentiate between real and fake about half of the time, hence the adversarial nature. A visual representation of the architecture can be seen in Figure 2.9.

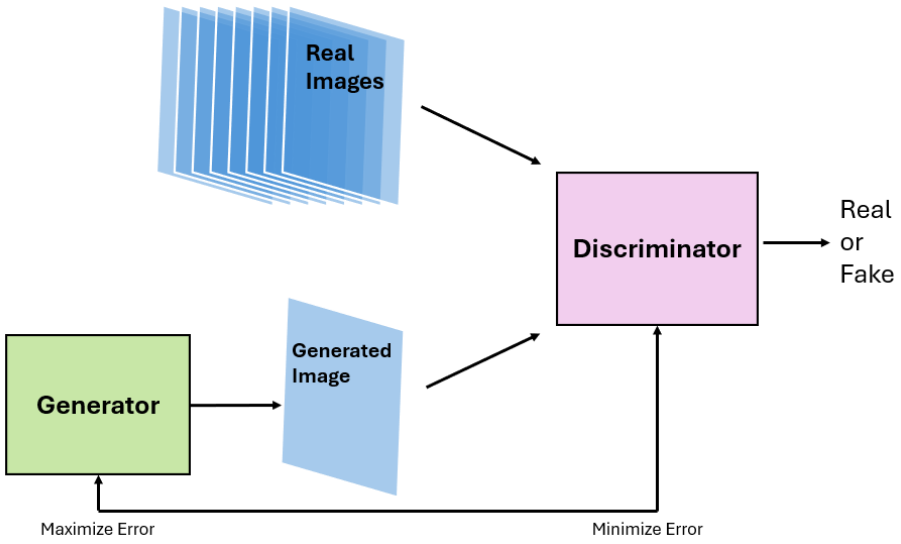


Figure 2.9: A visual representation of a GAN architecture.

2.3.1 GENERATOR

Generative models are not a new concept in the realm of machine learning and neural networks. A generative model is a form of unsupervised learning that is trained to generate new data. The generator of a GAN is a type of generative modeling. The purpose of the generator in the GAN framework is the learn the distribution of the inputted data and apply the learned information to generate new data [8]. The generator is fed noise (random data) and learns to create new images using the information gained from the discriminator.

The framework of the generator consists of an encoder, a decoder, and a connector between the two as seen in Figure 2.10 [13]. The encoder is a CNN, as described in Section 2.2. The purpose of the encoder is to decrease the size of the input image and learn the distribution of the data. This is done by decreasing the number of filters in each layer of the CNN [13]. On the other hand, the purpose of the decoder is to use the information learned in the encoding process to build/generate a new image. The decoder is a transpose CNN, a reverse version of a CNN where the dimensionality of the image is increased rather than decreased. To ensure this information is preserved, each layer of the decoder contains the output of the previous layer in the model as well as the input of the corresponding layer in the encoder [13]. This is represented in Figure 2.10, where the lighter color represents the layers of the decoder and the darker section is the input of the corresponding encoder layer.

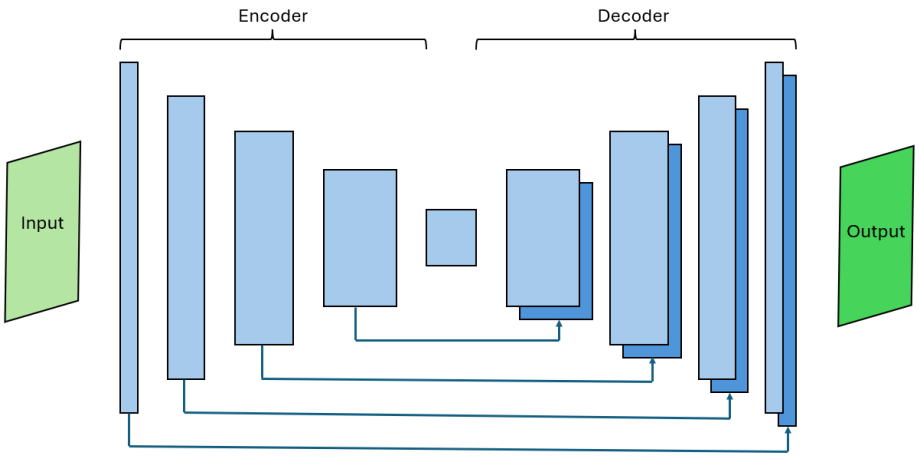


Figure 2.10: Generator architecture.

2.3.2 Discriminator

Classification networks are commonly referred to as discriminative models. Trained in competition with the generator, described in the above section 2.3.1, is the discriminator. It is a supervised model made up of a single convolutional neural network. It is designed to differentiate between a real image and a generated image. The discriminator is fed both real images and desired output images and is trained to differentiate between the two. Then images created by the generator are also inputted and the discriminator uses its learned information gained from the labeled data to try and classify it as real or fake.

2.3.3 Adversarial Training

A generative adversarial network is a new approach to generating images pins two models, a generator and a discriminator, against one another to produce new data similar to the input data. The adversarial nature is inspired by a two-person zero-sum game. In other words, the loss of the generator results in the equivalent gain of the discriminator and vice-versa. Generated images are inputted into the discriminator which then classifies the

image as a source image or a generated one. If the discriminator is correct, then it remains untouched. These results are then fed into the generator whose weights are then heavily altered to increase its ability to trick the discriminator. However, if the discriminator is wrong, the weights are updated to achieve a higher accuracy in its classification. The goal of the discriminator is to minimize its error (number of times it categorizes incorrectly), while the generator aims to maximize the same error.

CHAPTER 3

THE BEOWULF MANUSCRIPT

Beowulf is an ancient epic poem of great importance and renown. Written in Old English, the poem contains 3,183 alliterative lines and follows a Scandinavian hero named Beowulf [17]. It is not only considered one of the most significant pieces of Old English texts but also the most frequently translated. Although the dating of the manuscript is a controversial subject in humanities, many historians believe it was written sometime between 975 CE and 1025 CE. The only known copy of the poem is held in a manuscript known as the Nowell Codex. However, its official title is Beowulf manuscript (Cotton MS Vitellious) [11]. The title is inspired by one of the owners, Robert Bruce Cotton. In the 17th century, while under Cotton's ownership, it was held at the Ashburnham House in the Cotton Library alongside many other important manuscripts. In 1731, the Ashburnham House caught on fire. Although the Beowulf manuscript managed to survive, the edges and some of the text did suffer some damage [17].

The fire in the Cotton Library, a room housing many medieval manuscripts, is a huge motivating factor for providing digital support for the Beowulf manuscript and other manuscripts alike. While the Beowulf manuscript made it out of the fire with minimal damage, the same can not be said for many other manuscripts being held there.

3.1 Manuscript Images

The Beowulf manuscript was chosen as our model dataset because of multiple reasons: the manuscript images (Figure 3.1) are available online to the general public through an electronic edition (Figure 3.2), it is of great interest for humanities students and researchers, and it is used widely in classrooms. The problem of linking the manuscript images content and their corresponding text transcriptions has been of a large interest since the first electronic edition of Beowulf was published decades ago. Several attempts were performed

by manually recording metadata including image coordinates in the manuscript transcription. Such approach is tedious and unlikely to produce good searching capabilities for manuscript images.

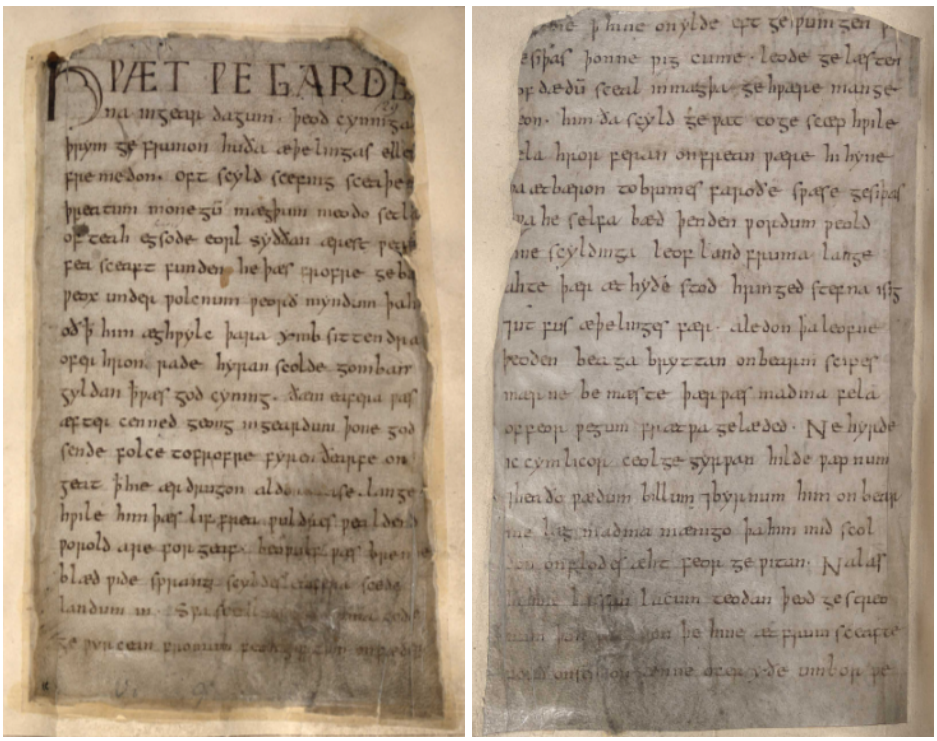


Figure 3.1: First two pages from the electronic images of the Beowulf manuscript

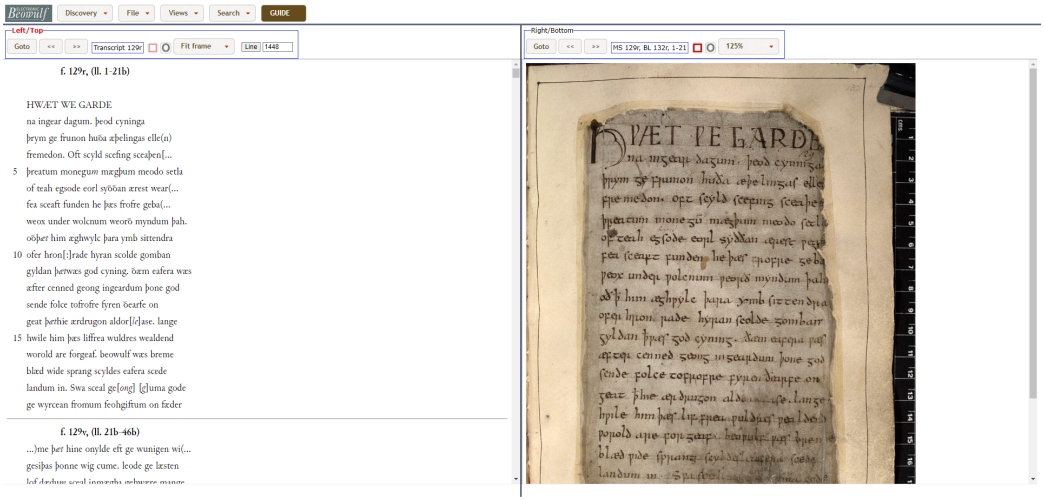


Figure 3.2: The Electronic Beowulf

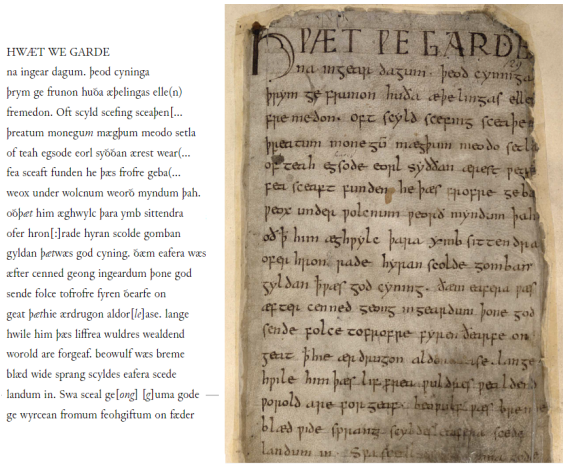
With the impressive advances in Artificial Intelligence and Machine Learning techniques and support in the past decade, the idea of automating the process of linking manuscript images and their corresponding text transcriptions seems more realistic than ever. Our current work builds on previous attempts to perform automatic individual manuscript character recognition and sets the goal of producing character recognition for a whole manuscript page at the time. We create a model, based on Generative Adversarial Networks architecture, that we trained using a reduced set of original manuscript images and snapshot images of their transcriptions (from the electronic edition in Figure 3.2), and attempt to generate new transcription images from any manuscript image. While our method is not yet perfect, it lays the foundation for producing models capable of recognizing languages other then Old English from their original manuscript images.

In the subsequent section we describe how we prepare our dataset of images for training the model, which is described in Chapter 4.

3.2 Preprocessing Image Data

For our model we selected 25 pairs of manuscript and transcription images as shown in Figure 3.3 and each image was scaled to the size 512×256. Neural Network based models success is typically linked to the amount of data used for training, and, in the case of models that process images, to the size of each image in the dataset. As one of our goals was to create a model that uses a relatively small dataset for training, we organized our data and experiments in three ways.

We first used the original dataset of all 25 pairs of images and designed a model that trains with images of a full manuscript page size. The images in this dataset category are illustrated in Figure 3.3.



HWÆT WE GARDE
na ingear dagum. þeod cyninga
þrym ge frunon huða æþelingas elle(n)
fremedon. Oft scyld scefing sceaþen[...
þreatum monegum mægþum meodo setla
of teah egsode eorl syððan ærest wear(...
fea sceaft funden he þæs frofre geba(...
weox under wolcnum weorð myndum þah.
oðþæt him æghwylc þara ymb sittendra
ofer hron[:]rade hyran scolde gomban
gyldan þætwæs god cyning. ðæm eafera wæs
æfter cenned geong ingeardum þone god
sende folce tofrofre fyren ðearfe on
geat þærhie ærdrugon aldor[le]ase. lange
hwile him þæs liffrea wuldres wealdend
worold are forgeaf. beowulf wæs breme
blæd wide sprang scyldes eafera scede
landum in. Swa sceal ge[ong] [g]uma gode —
ge wyrcean fromum feohgiftum on fæder

Figure 3.3: The Electronic Beowulf manuscript and transcription images

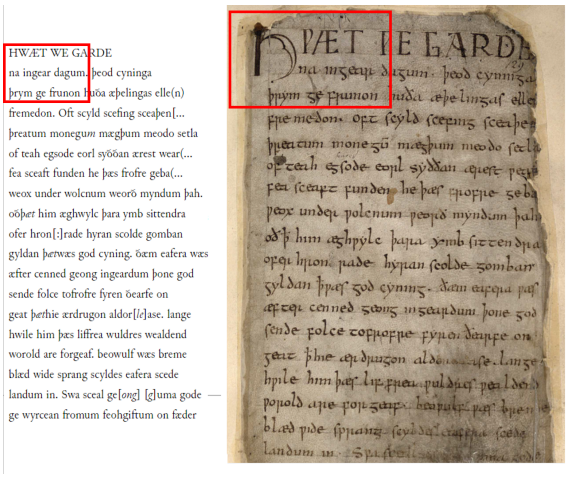


Figure 3.4: The Electronic Beowulf manuscript and transcription images slicing (1)

Next, we automatically created significantly larger datasets of images by considering subsets of manuscript pages. More specifically, we extracted image fragments by taking a sliding window moving from the top of each image to the left and down the page. We experimented with a few sliding window sizes and horizontal/vertical steps, with overlapping between consecutive image fragments. The process of extracting image fragments is illustrated in Figures 3.4-3.6.

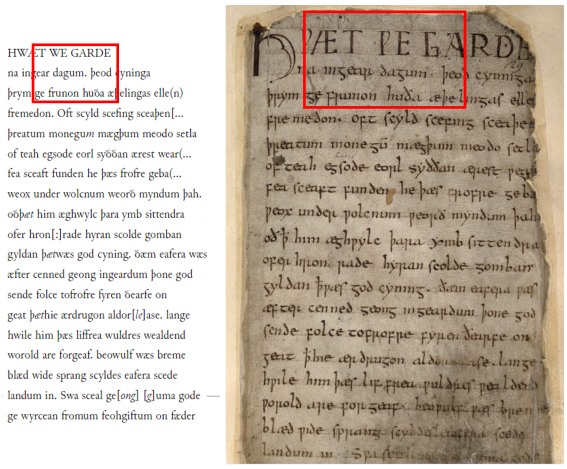


Figure 3.5: The Electronic Beowulf manuscript and transcription images slicing (2)

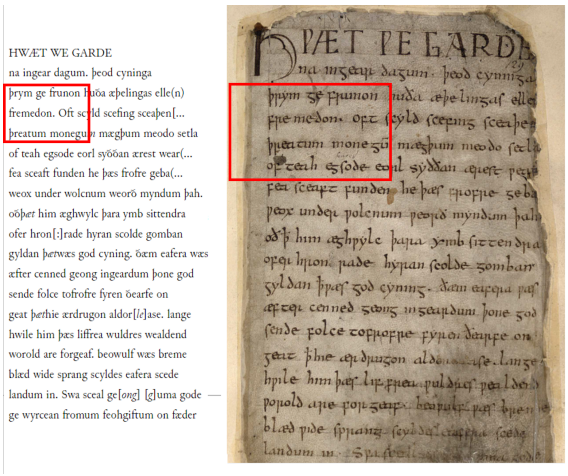


Figure 3.6: The Electronic Beowulf manuscript and transcription images slicing (3)

We have chosen on purpose to have overlapping between consecutive image fragments, as same words and characters appearing in multiple fragments was expected to improve the model's training. However, the method introduces two important limitations. Firstly, each image fragment cuts words and characters on both vertical and horizontal edges. Secondly, there is a clear dis-alignment between content of a manuscript image fragment and transcription image fragment, which inherently happens due to different character shapes and sizes in the two images. We found this second limitation more disturbing, as it was more likely to negatively affect the model's training.

Finally, we created larger dataset of image fragments by using a vertical sliding window of same width as the original images (that is, width 256). With this dataset we partially avoided some of the limitations listed above, with the dia-alignment almost completely avoided. The downside was a smaller set of images for model's training.

All experimental results are reported in Chapter 5.

CHAPTER 4

A GAN MODEL FOR MANUSCRIPT IMAGE TRANSLATION

We present a Generative Adversarial Network based model for converting manuscript images into their corresponding, clear images of the manuscript text. The chapter is organized as follows. We provide background and notations in Section 4.1. Our model for converting manuscript images to legible images of the content is introduced in Section 4.2. The corresponding mathematical model is described in Section 4.3, then Section 4.4 describes the training process and evaluation for the model.

4.1 BACKGROUND AND NOTATIONS

The main goal of the current project is to produce a model capable of taking a snapshot image from an old manuscript book page (the source image) and converting it into a snapshot image (the target image) of the corresponding typed manuscript text, as shown in Figure 4.1

The input and output of our model consist of images, which are commonly represented as embedded 3D real valued functions $f(x,y,z) \mapsto \mathscr{I} \in \mathbb{R}^{H\times W\times C}$, for some integers H, W, and C which represent the height, width, and number of channels, respectively. The function $f(x,y,z)$ produces the value of the pixel at position (x,y,z) in the image. The representation $\mathscr{I}$ is a 3D array, which is informally referred as "data tensor", however, the concept of a "tensor" in the strict mathematical formulation is a generalization of a linear mapping $f : V_1 \times \cdots \times V_n \to W$ (a multilinear mapping), where $V_1, \ldots, V_n, W$ are vector spaces. Data tensor representation benefits from significant theoretical support for operating with tensors allowing various complex observations (images, sounds, 3D objects, etc.) be analyzed using specific tensor methods [16, 15, 14]. For an image representation $\mathscr{I} \in \mathbb{R}^{H\times W\times C}$, we denote $dim(\mathscr{I}) = (H,W,C)$ the dimension of the image, representing the height, width, and number of channels, respectively. We note that a typical image has either $C = 3$ (RGB

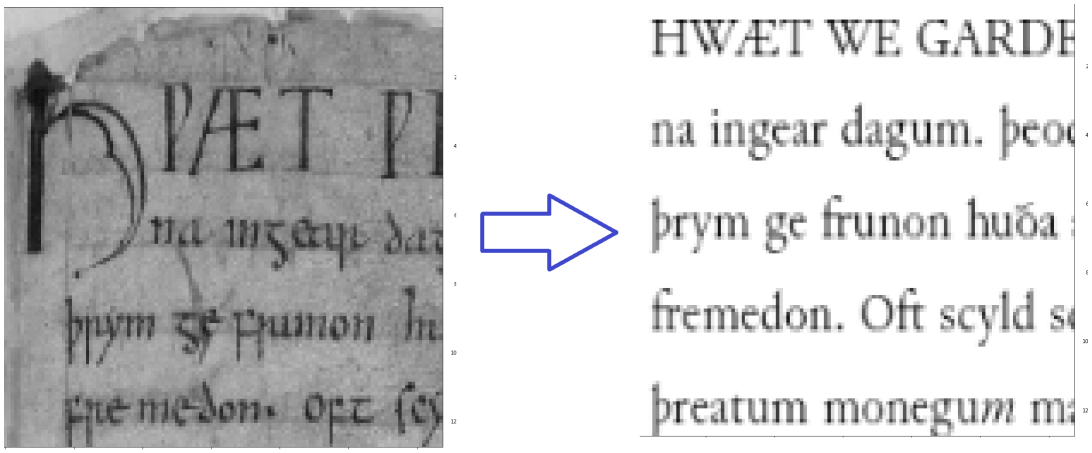


Figure 4.1: Manuscript image conversion to typed text image

representations) or $C = 1$ (BW or grayscale representations). All our source images will be represented as grayscale image, hence $C = 1$. However, in the process of converting a source image to a target image the representation $\mathscr{I}$ may be successively converted into a sequence of multidimensional arrays of various heights, widths, and number of channels.

The parametric image translation model we propose can be formally defined as follows.

Definition 4. An image translation model is a mapping

$$\mathscr{M} : \mathbb{R}^{H_i \times W_i \times C_i} \to \mathbb{R}^{H_o \times W_o \times C_o} \quad \mathscr{M}(\mathscr{I}; \mathbf{W}) = \mathscr{T}$$

which takes an input image $\mathscr{I}$ of dimension (H_i, W_i, C_i) and produces a target image $\mathscr{T}$ of dimension (H_o, W_o, C_o), where $\mathbf{W}$ represents the set of model's parameters.

The image translation is performed by the model using successive transformations (also called "layers"), with each such transformation taking as input an image representation and producing as output another image representation. Moreover, each such intermediate transformation can use parameters or be parameter-less transformation. The union of all intermediate transformations parameters produce the set $\mathbf{W}$ of parameters of the model $\mathscr{M}$.

Each layer taken individually is a mini-model that performs a single image transformation, as formally defined next.

Definition 5. An image transformation layer is a mapping

$$\mathscr{L} : \mathbb{R}^{H_i \times W_i \times C_i} \to \mathbb{R}^{H_o \times W_o \times C_o} \quad \mathscr{L}(\mathscr{I};\, \mathbf{W}_l) = \mathscr{T}$$

which takes an input image $\mathscr{I}$ of dimension (H_i, W_i, C_i) and produces a target image $\mathscr{T}$ of dimension (H_o, W_o, C_o), where $\mathbf{W}_l$ represents the set of layer's parameters.

However, the power of the model $\mathscr{M}$ relies in combining multiple layers together and optimizing their parameters to produce a translation as accurate as possible. Multiple image transformation layers are combined in the final model as follows:

$$\begin{aligned} \mathscr{M}(\mathscr{I};\mathbf{W}) &= (\mathscr{L}_n \circ \mathscr{L}_{n-1} \circ \cdots \circ \mathscr{L}_1)(\mathscr{I}) \\ \mathbf{W} &= \bigcup_{i=1}^{n} \mathbf{W}_i \end{aligned} \tag{4.1}$$

where $\mathbf{W}_i$ is the set of parameters for layer $\mathscr{L}_i$ and n is the number of layers.

The number n of layers and their types are typically established by practical trials. The optimal values of the model parameters are determined as solutions of an optimization problem, which will be described in the subsequent sections.

For our model $\mathscr{M}$ we used four types of image transformation layers (2D convolutional, batch normalization, activation, and transpose 2D convolutional layer), which will be formally described in the following section. We must also note that we only present the layers used in our model, whereas in practice there are some other available transformation layers.

4.2 IMAGE TRANSFORMATION LAYERS

4.2.1 THE 2D CONVOLUTION AND TRANSPOSE 2D CONVOLUTION LAYERS

As informally described in Section 2.2.1, the 2D convolution layers are the backbone of a Convolutional Neural Network (CNN). They typically covert an input image (represented as a multidimensional array) $\mathscr{I}(H_i, W_i, C_i)$ into an output multidimensional array image $\mathscr{O}(H_o, W_o, C_o)$ of larger dimensions $C_i < C_0$. Each slice of the output image represents a version of the input image where one or more features are enhanced (typically edges in one or more directions). The convolution layer performs this transformation by means of filters and strides. A filter is a set of small matrix of weights (typically squared matrices 3×3, 4×4, 5×5, etc.) that is applied to each image (array slice) in $\mathscr{I}(H_i, W_i, C_i)$ to produce a new image with a certain enhanced feature (the process is called *feature extraction*). One matrix of parameters in a filter is called a kernel and a filter is composed of C_i kernels for an input image $\mathscr{I}(H_i, W_i, C_i)$. In Section 2.2.1 we show an example of applying a filter to an input image. In summary, a filter feature extraction for one 2D image transforms every entry (pixel) of the input image into another image where each entry (pixel) is a weighted average of the corresponding input image pixel and its neighbors covered by the kernel size. However, such transformation does not necessary produce same size output image as the input image due to the fact the kernel is applied to input matrix pixels that are s units apart, rather then to every pixel. The value s is called *stride* and may have distinct values for moving horizontally and vertically. Consequently, the output image dimensions are scaled down by the s value. The image transformation performed by a filter (called feature extraction) is formally described as follows.

Definition 6. Let $\mathscr{I}(H_i, W_i, C_i)$ be a representation of an image and $\mathscr{F} = \{K_1, \ldots, K_{C_i}\}$ a filter, with kernels $K_k \in M^{f_1, f_2}$, $K_k = [w_{ij}^{(k)}]$, $k = 1 \ldots C_i$. f_1, f_2 is the kernel dimension and $w_{ij}^{(k)}$ are the weights associated to the kernel. Let s be the stride associate to the transfor-

mation.

The feature extraction performed by $\mathscr{F}$ on image $\mathscr{I}(H_i, W_i, C_i)$ is a transformation

$$\mathscr{F} : \mathbb{R}^{H_i \times W_i \times C_i} \to \mathbb{R}^{H_o \times W_o \times C_o}$$

$$\mathscr{F}(\mathscr{I}) = \mathscr{O},\ \mathscr{O} = [o_{ij}, k]$$

$$o_{ij} = \mathscr{I}[i' - \lceil f_1/2 \rceil : i' + \lceil f_1/2 \rceil, j' - \lceil f_2/2 \rceil : j' + \lceil f_2/2 \rceil, k] \odot K_k + b_k,\ k = 1 \ldots C_i$$

where $\odot$ denotes the Hadamard (element-wise) product of two matrices, b_k is a trainable parameter called bias, and by $i - \lceil f_1/2 \rceil : i + \lceil f_1/2 \rceil, j - \lceil f_2/2 \rceil : j + \lceil f_2/2 \rceil$ we denote a selection from a matrix in *from:to* format. The pixels positions (i, j) in the output matrix and (i', j') in the input matrix are related by the stride s factor: $i' = i \cdot s,\ j' = j \cdot s$.

This formal definition of a filter transformation (filter feature extraction) will allow us to perform a rigorous count of the model parameters and the output image dimensions. For an input image $\mathscr{I}(H_i, W_i, C_i)$ and a filter transformation with stride s, the output image dimension is $(\lfloor H_1/s \rfloor, \lfloor W_i/s \rfloor)$, that is, the output image is scaled down by the factor s.

A 2D convolution layer is a collection of filters applied to a multidimensional input image, as described next.

Definition 7. Let $\mathscr{I}(H_i, W_i, C_i)$ be an input image and $\mathscr{F} = \{F_1, \ldots, F_f\}$ a set of f filters with stride s. A 2D convolution layer is a transformation

$$Conv2D : \mathbb{R}^{H_i \times W_i \times C_i} \to \mathbb{R}^{H_o \times W_o \times C_o}$$

$$Conv2D(\mathscr{I}) = \mathscr{O}$$

where $H_o = \lfloor H_i/s \rfloor$, $W_o = \lfloor W_i/s \rfloor$, $C_o = f \cdot C_i$, and $\mathscr{O} = [F_1(\mathscr{I}), \ldots, F_f(\mathscr{I})]$.

That is, a 2D convolution layer applies each filter in the set of filters to the input image representation and stacks the results in the output image representation. In a 2D convolutional layer the number f of filters, the size of each kernel $f_1 \times f_2$ and the strid are

chosen based on experimental trials, whereas the weights $w_{ij}^{(k)}$ for each kernel in each filter are parameters to be determined through an optimization process for the model (trainable parameters). Consequently, for a 2D convolutional layer with f filters and kernels of size $f_1 \times f_2$, the set of trainable parameters is

$$\begin{aligned} \mathbf{W} &= \bigcup_{filter\ F} \bigcup_{K_k \in F} \{w_{ij}^{(k)}, b_k\} \\ |\mathbf{W}| &= f \cdot C_i \cdot (f_1 \cdot f_2 + 1) \end{aligned} \tag{4.2}$$

The process of finding optimal trainable parameters of the model will be described in the upcoming sections.

A transpose 2D convolutional model is the inverse of a 2D convolution transformation. It uses filters and stride (with same number of trainable parameters) to perform an inverse transformation of an input image. The only difference is that the dimension of the output image is being stretched by the stride value s.

4.2.2 The batch normalization layer

A batch normalization layer performs statistical adjustments of values in an input image representation for practical purposes: typically the process of finding all optimal model parameters is more effective (faster and more stable). The numerical adjustments are performed based on statistics (mean and standard deviations) over a set of input images, called a batch. For such a batch, the values of mean μ_c and standard deviation σ_c are computed and stored for each channel slice (matrix), then the input image representation is processed as described below.

Definition 8. A batch normalization transformation is a mapping

$$BN : \mathscr{I}(H_i, W_i, C_i) \to \mathscr{O}(H_o, W_o, C_o), \;\; BN(\mathscr{I}) = \mathscr{O}$$

where $H_o = H_i = H$, $W_o = W_i = W$, $C_o = C_i = C$ and $\mathscr{O}[i,j,c] = \gamma_c \cdot \dfrac{\mathscr{I}[i,j,c] - \mu_c}{\sigma_c} + \beta_c$, $i = 1,\ldots,H$, $j = 1,\ldots,W$, $c = 1,\ldots,C$. The parameters γ_c and β_c are trainable parameters and are optimally determined during the model optimization process.

Therefore, the parameters stored by the model for a batch normalization layer that performs an image transformation $\mathbb{R}^{H\times W\times C} \to \mathbb{R}^{H\times W\times C}$ are:

$$\begin{aligned} \mathbf{W} &= \{\gamma_c, \beta_c, \mu_c, \sigma_c\}, \ \ c = 1,\ldots,C \\ |\mathbf{W}| &= 4 \cdot C \end{aligned} \tag{4.3}$$

4.2.3 The activation layer

The activation layer performs an entry-wise non-linear transformation on each entry of the input image. Other than choosing the specific non-linear function (which is called activation function), the layer has no parameters.

Definition 9. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, increasing function. The activation layer is a transformation

$$A : \mathbb{R}^{H_i\times W_i\times C_i} \to \mathbb{R}^{H_o\times W_o\times C_o}, \ \ A(\mathscr{I}) = \mathscr{O}$$

where $H_o = H_i = H$, $W_o = W_i = W$, $C_o = C_i = C$ and $\mathscr{O}[i,j,c] = \sigma(\mathscr{I}[i,j,c])$.

Some popular choices for the activation function are: the sigmoid function, hyperbolic tangent, RELU, or leaky-RELU.

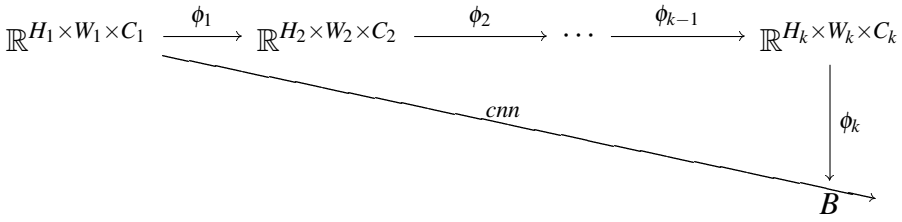
- Sigmoid function: $\sigma(x) = \dfrac{1}{1+e^{-x}}$
- Hyperbolic tangent: $tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
- Rectified Linear Activation Function (RELU): $f(x) = max(0,x)$
- Leaky-RELU: $f(x) = max(\alpha x, x)$, where $0 < \alpha \ll 1$.

The activation layer does not add any parameter to the model.

4.3 CONVOLUTIONAL NEURAL NETWORKS MODELS

A Convolution Neural Network model (informally introduced in Section 2.2) can be formally described by a commutative diagram of combining various image transformation layers.

Definition 10. A Convolutional Neural Network (CNN) is a composition $cnn : \mathbb{R}^{H\times W\times C} \rightarrow B$ of image transformation layers $\mathbb{R}^{H_1\times W_1\times C_1} \xrightarrow{\phi_1} \mathbb{R}^{H_2\times W_2\times C_2} \xrightarrow{\phi_2} \ldots \xrightarrow{\phi_k} B$ described by the commutative diagram:



where each $\phi_j,\ j = 1...k$ is an image transformation layer and the codomain B is either a multidimensional image array $\mathbb{R}^{H\times W\times C}$ or a discrete set of L labels $\{1,2,\ldots,L\}$.

The typical purpose of a CNN model is to perform image encoding/transformation (in which case $B = \mathbb{R}^{H\times W\times C}$) or an image classification (in which case $B = \{1,3,\ldots,L\}$). In our model, we use two CNN-based models, one models for each scenario. The parameters set for a CNN model is the union of parameters sets for each transformation layer in its composition:

$$W_{cnn} = \bigcup W_k, \text{ where } W_k \text{ is the parameters set of } \phi_k.$$

4.4 IMAGE TRANSLATION MODELS USING GENERATIVE ADVERSARIAL NETWORKS MODELS

A Generative Adversarial Network (GAN) is a framework introduced by Ian Goodfellow in 2014 [7] and it typically consists of two Artificial Neural Network models competing

against each-other on generating artificial (fake) information on one hand, and distinguishing between artificially created (fake) information and real information, on the other hand. The two ANN-based components are called the Generator and the Discriminator, respectively.

Our image translation model based on Generative Adversarial Networks is sketched in Figure 4.2.

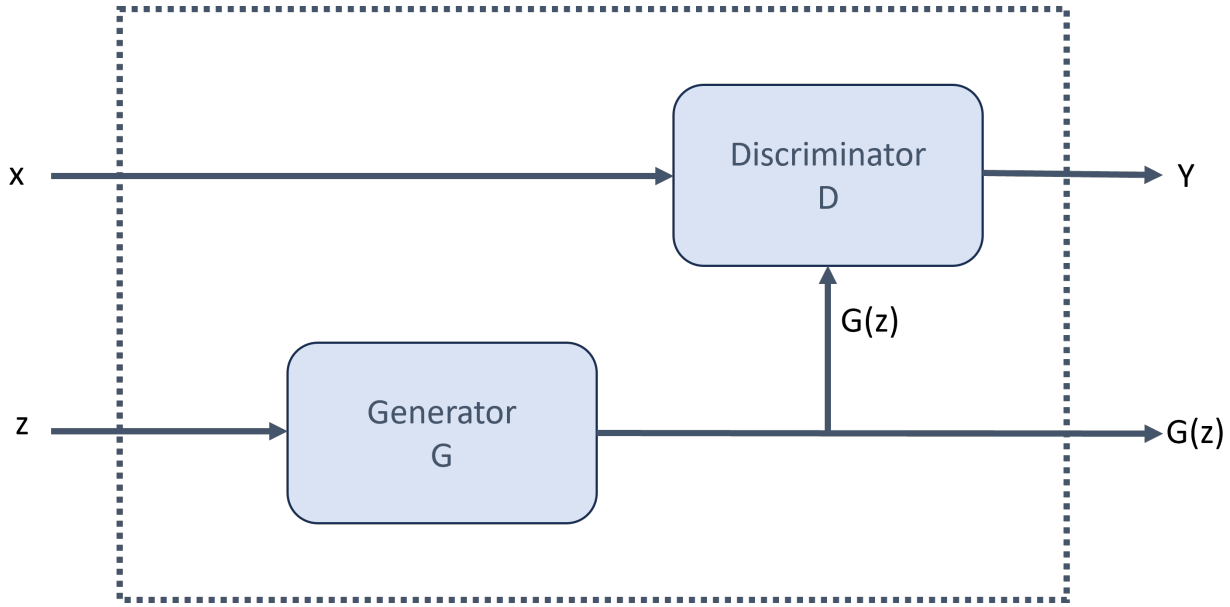


Figure 4.2: A GAN model for image translation

In the GAN model represented in Figure 4.2 the generator G is a function that takes artificial information z as input and aims to produce credible information $G(Z)$ as the output. The generator's output is provided to the discriminator D, which is a model design to take an input x and produce an output $D(x) = y \in \{True, False\}$ which must distinguish as accurately as possible between real information x and artificial (fake) information $G(z)$. Both G and D are types of ANN-based models with a very large set of parameters. The GAN model learns the optimal parameters (for both G and D) to produce the desired results through a process called *training*.

A formal introduction of the GAN model and how the model computes its optimal parameters are provided in the subsequent subsections.

4.4.1 THE MODEL LOSS FUNCTIONS

Let $\mathscr{I} = \mathscr{I}_s \cup \mathscr{I}_t$ be a set of input images partitioned into two disjoint, equal cardinality subsets $\mathscr{I}_s$ of source images and $\mathscr{I}_t$ of target images. We let the bijection $o : \mathscr{I}_s \to \mathscr{I}_t$ be the image translation model objective transformation function. We definite the GAN model loss functions as follows.

Definition 11. [Discriminator model loss] The Discriminator model loss function is a real valued function $L_D : 2^{\mathscr{I}_s} \to \mathbb{R}$,

$$L_D(I) = \sum_{i \in I} |D(i) - 1|^2 + \sum_{i \in o(I)} |D(i) - 1|^2 + \sum_{i \in I} |D(G(i))|^2$$

Definition 12. [Generator model loss] The Generator model loss function is a real valued function $L_G : 2^{\mathscr{I}_s} \to \mathbb{R}$,

$$L_G(I) = \sum_{i \in I} |]G(i) - o(i)||^2$$

Definition 13. [GAN model loss] The GAN model loss function is a real valued function $L : 2^{\mathscr{I}_s} \to \mathbb{R}$,

$$L(I) = L_G(I) + \sum_{i \in I} |D(i) - 1|^2 + \sum_{i \in I} |D(G(i))|^2$$

4.4.2 TRAINING AND EVALUATING THE MODEL

Algorithm 1 describes the training process for the GAN-based model for image translation.

Algorithm 1 The GAN model training algorithm

Input: model M with parameters W, $\mathscr{I}_s = \mathscr{I}_1 \cup \mathscr{I}_2 \cup \cdots \cup \mathscr{I}_B$, $EPOCHS$

Output: model M with optimal parameters W

$W \leftarrow random()$

for $i = 1, \ldots, EPOCHS$ **do**

    for $k = 1, 2, \ldots, B$ **do**

        $\min_W L_D(\mathscr{I}_k)$

        Adjust Discriminator model parameters

        $\min_W L(\mathscr{I}_k)$

        Adjust Discriminator and Generator models parameters

    end for

end for

Print $L_D(\mathscr{I})$, $L_G(\mathscr{I})$

CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented our models in Python using Tensorflow [1] and Keras [4] libraries for implementing Artificial Neural Network based models. Our tests were initially ran on a PC Desktop computer using an Intel(R) Core(TM) i7-10700K CPU 3.80GHz processor equipped with 16GB of RAM. The initial experiments were taking between 12-72 hours to complete. We subsequently switched to the Talon High Performance Cluster (HPC) computer system at Georgia Southern University and were able to run complete experiments in about 8-24 hours per experiment.

The models implementation and experimental results are reported in the subsequent sections.

5.1 MODEL IMPLEMENTATION

The Python implementations of the Discriminator, Generator, and the GAN models are presented in Listings 5.1, 5.2, and 5.3, respectively.

```
def define_discriminator(image_shape):
  # weight initialization, source and target inputs and merge them
  init = RandomNormal(stddev=0.02)
  in_source_image = Input(shape=image_shape)
  in_target_image = Input(shape=image_shape)
  merged = Concatenate()([in_source_image, in_target_image])
  # CNN: 64 filters
  hl = Conv2D(64, (3,3), strides=(2,2), padding='same',
   kernel_initializer=init)(merged)
  # set alpha to 0.2 rather than the default 0.3 to account for some
   negative pixels
  hl = LeakyReLU(alpha=0.2)(hl)
  # CNN: 128 filters
```

```
12  hl = Conv2D(128, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
13  # BatchNormalization so that the inputs are standardized (mean = 0,
     std  = 1)
14  hl = BatchNormalization()(hl)
15  hl = LeakyReLU(alpha=0.2)(hl)
16  # CNN: 256 filters
17  hl = Conv2D(256, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
18  hl = BatchNormalization()(hl)
19  hl = LeakyReLU(alpha=0.2)(hl)
20  # CNN: 512 filters
21  hl = Conv2D(512, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
22  hl = BatchNormalization()(hl)
23  hl = LeakyReLU(alpha=0.2)(hl)
24  # 2nd to last output layer
25  hl = Conv2D(512, (3,3), padding='same', kernel_initializer=init)(hl)
26  hl = BatchNormalization()(hl)
27  hl = LeakyReLU(alpha=0.2)(hl)
28  # output layer
29  hl = Conv2D(1, (3,3), padding='same', kernel_initializer=init)(hl)
30  patch_out = Activation('sigmoid')(hl)
31  # define and compile model
32  model = Model([in_source_image, in_target_image], patch_out)
33  opt = Adam(learning_rate=0.0002, beta_1=0.5)
34  model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights
     =[0.5])
35  return model
```

Listing 5.1: The Discriminator model implementation

```
1 def encoder_block(layer_in, n_filters, batchnorm=True):
```

```
  init = RandomNormal(stddev=0.02)
  g = Conv2D(n_filters, (3,3), strides =(2,2), padding='same',
   kernel_initializer=init)(layer_in)
  # Batch normalization and ReLU functions
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  g = LeakyReLU(alpha=0.2)(g)
  return g

def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  init = RandomNormal(stddev=0.02)
  g = Conv2DTranspose(n_filters, (3,3), strides=(2,2), padding='same',
   kernel_initializer=init)(layer_in)
  g = BatchNormalization()(g, training=True)
  # Add a drop out to the earlier layers of the decoding part of the
   generator.
  # This is 'cheap way' to regularize the deep neural network.
  # We drop out half of the input variables from the previous layer.
  if dropout:
    g = Dropout(0.5)(g, training=True)
  # merge with output of corresponding contracting block
  #g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g

def define_generator(image_shape=(IMG_FRAG_HEIGHT, IMG_FRAG_WIDTH, 1)):
  init = RandomNormal(stddev=0.02)
  in_image = Input(shape=image_shape)
  # encoder model
  e1 = encoder_block(in_image, 64, batchnorm=False)
```

```
  e2 = encoder_block(e1, 128)
  e3 = encoder_block(e2, 256)
  e4 = encoder_block(e3, 512)
  #e5 = encoder_block(e4, 512)
  #e6 = encoder_block(e5, 512)
  #e7 = encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (3,3), strides=(2,2), padding='same',
   kernel_initializer=init)(e4)
  b = Activation('relu')(b)
  # decoder model
  #d1 = decoder_block(b, e7, 512)
  #d2 = decoder_block(d1, e6, 512)
  #d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(b, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output and model
  g = Conv2DTranspose(1, (3,3), strides=(2,2), padding='same',
   kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)
  model = Model(in_image, out_image)
  return model
```

Listing 5.2: The Generator model implementation

```
def define_gan(g_model, d_model, image_shape):
  in_source = Input(shape=image_shape)
  # discriminator not trainable
  d_model.trainable = False
  # generate image from in_source
  gen_out = g_model(in_source)
```

```
    # pass both in_source and gen_out to discriminator
    dis_out = d_model([in_source, gen_out])
    # build a model where image source is compiled against both
     descrimination and generated image
    model = Model(in_source, [dis_out, gen_out])
    opt = Adam(learning_rate=0.0002, beta_1=0.5)
    model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt,
     loss_weights=[1,100])
    return model
```

Listing 5.3: The GAN model implementation

```
def train(d_model, g_model, gan_model, dataset, n_epochs=500, n_batch=3)
    :
    # dataset
    patch_shape1 = d_model.output_shape[1]
    patch_shape2 = d_model.output_shape[2]
    trainA, trainB = dataset
    # the epochs, the batch per epoch and the total number of training
     iterations
    bat_per_epo = int(len(trainA) / n_batch)
    n_steps = bat_per_epo * n_epochs
    for i in range(n_steps):
      # select a batch of real samples and update discriminator
      [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch,
       patch_shape1, patch_shape2)
      d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
      # generate a batch of fake samples and update discriminator
      X_fakeB, y_fake = generate_fake_samples(g_model, X_realA,
     patch_shape1, patch_shape2)
      d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
      # update the generator and summarize performance
      g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
```

```
18    print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i+1, d_loss1, d_loss2,
    g_loss))
19        #summarize every 100 epochs and at the last step
20    if ((i+1) == n_steps) or ((i+1) % (bat_per_epo*100) == 0):
21      summarize_performance(i, g_model, dataset)#, save_model=((i+1) ==
    n_steps))
```

Listing 5.4: Training the GAN model

The complete listing of the Python code can be found in Appendix A.

5.2 Experimental results for processing fragments of manuscript pages

We ran our experiments for various fragments shapes and sizes out of a manuscript page. The results presented here are produced from using square image fragments of 64×64 pixels. We produced these fragments by sliding a 64×64 window over the manuscript page, moving left to right and top to bottom using a 32 pixels step. The technique has the advantage of producing more source images for training (out of a limited number of manuscript pages), but the disadvantage of misalignments between the original image and the corresponding transcription snapshots (as characters have different widths in the two representations.

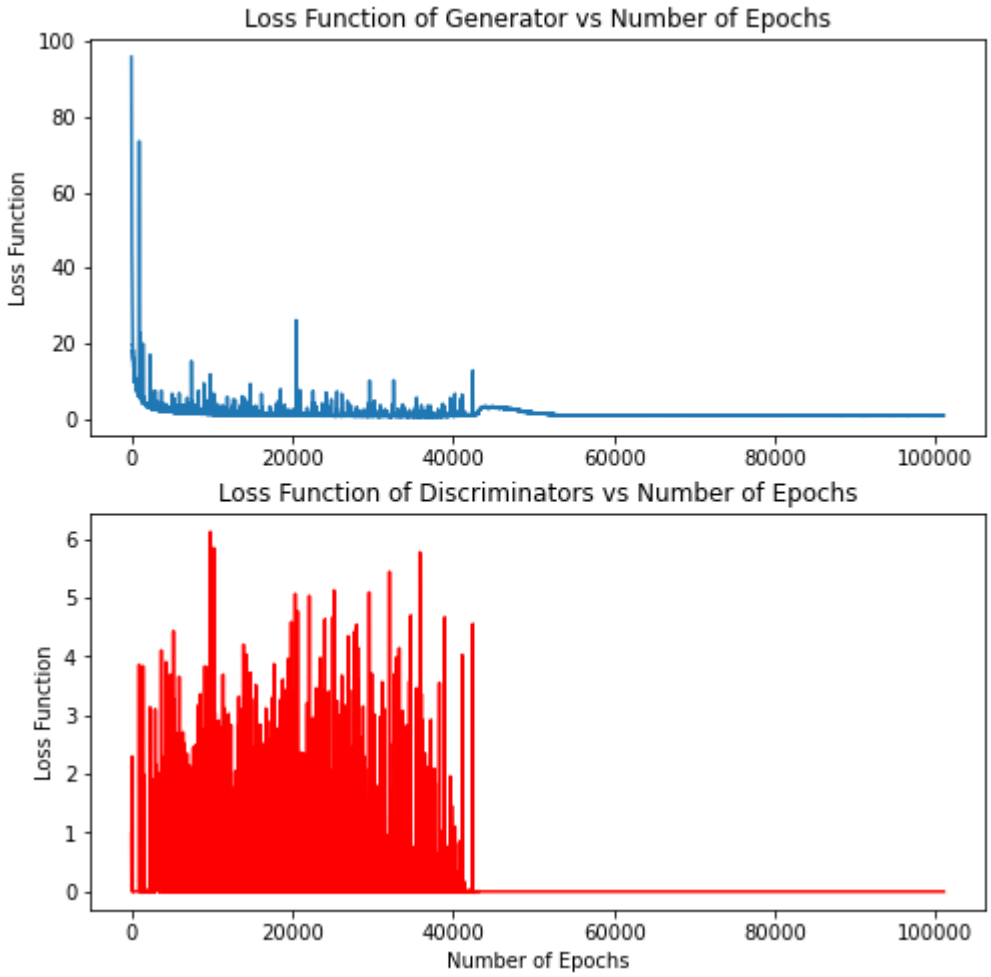


Figure 5.1: The Loss function for training the GAN model on image fragments

Figure 5.1 shows the Loss functions (discriminator and the entire GAN model) values during the training process of 100,000 epochs. As the curves flatten long before the end of the process, the models are completely trained.

Figure 5.2: Output of GAN model for sample training data: original manuscript image fragment (top), target image (bottom), and model output image (middle)

Figure 5.2 shows a pair of randomly chosen manuscript image fragments (on top), their corresponding transcription snapshots (bottom), and the generated versions of the transcriptions (middle). One can easily notice the misalignments between the characters in the original manuscript snapshots (top) and their translations (bottom). This is a clear limitation of this approach, which we tried to overcome by processing whole manuscript pages (the results presented in the subsequent section).

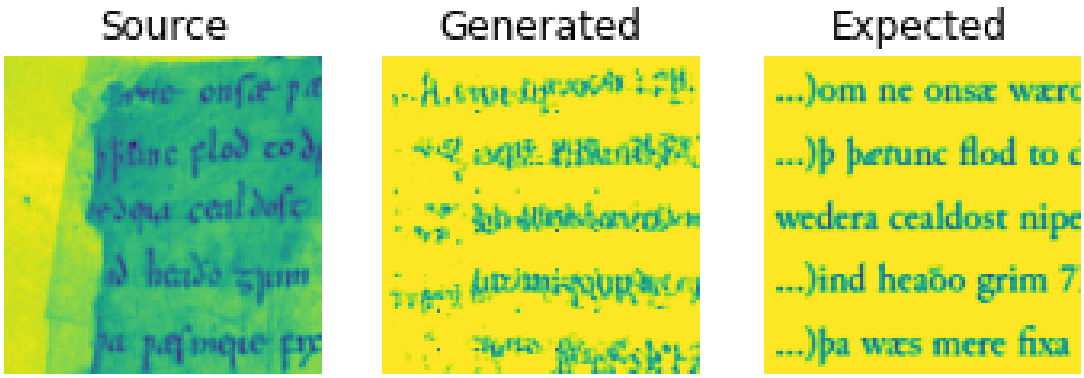


Figure 5.3: The GAN model output on a random image fragment input

Finally, Figure 5.3 shows the triplet of original image (left), original transcription snapshot (right), and the generated transcription image (middle).

One special experiment case scenario is the case of fragment images with full image width. We performed the next set of experiments with fragment images of dimension 64×256 (matching the full width of a manuscript image, sliding down 32 pixels. The purpose of the experiment was to determine whether the problem of misalignment (described in Chapter 3) will be alleviated.

Figure 5.4 shows the loss function during the training process. The figure shows that the experiment would need considerably more epochs for training the discriminator model. The results of generating random images from the training set are presented in Figure 5.5, and there generated image from the testing set is shown in Figure 5.6. While not convincing, we plan to repeat the experiment in future work using better training.

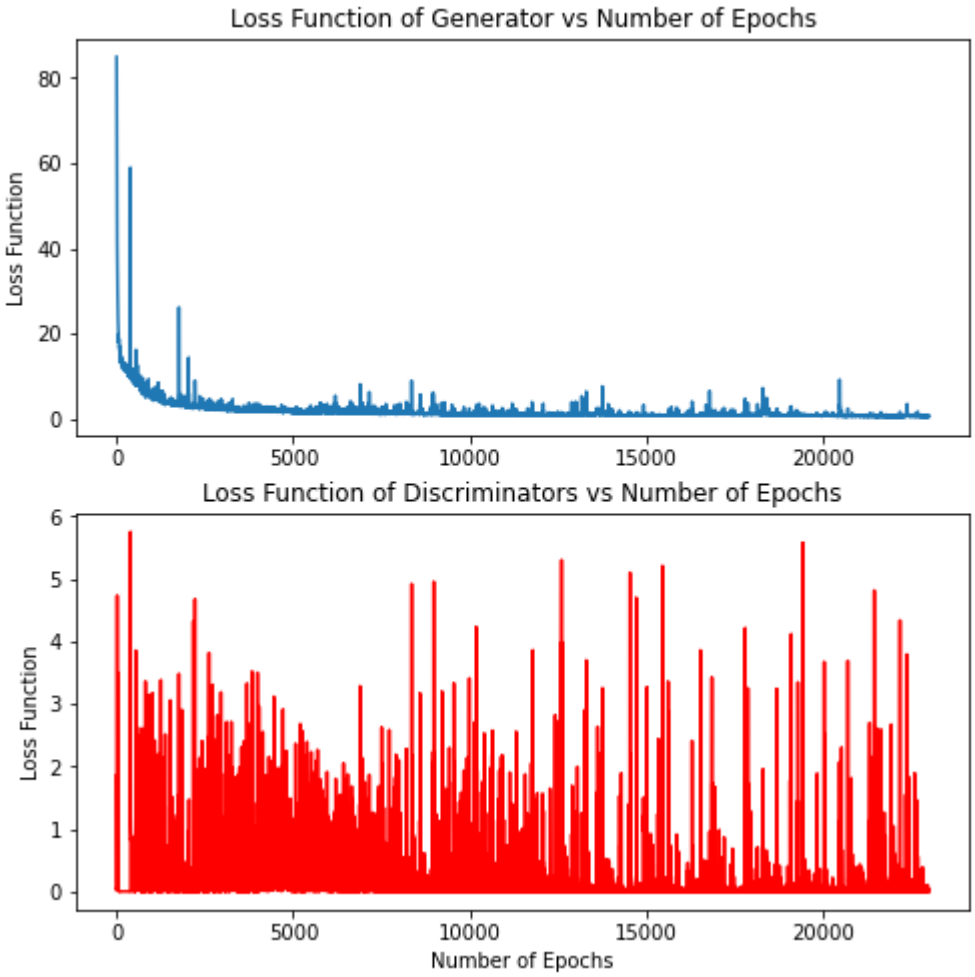


Figure 5.4: The Loss function for training the GAN model on image fragments

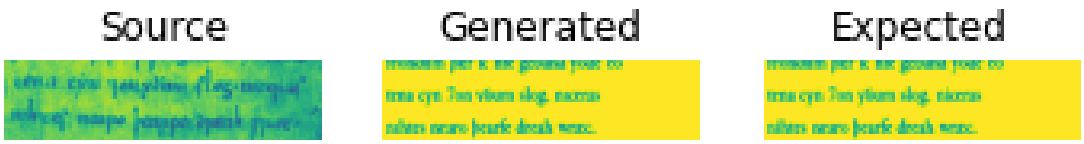


Figure 5.5: Output of GAN model for sample training data: original manuscript image fragment (top), target image (bottom), and model output image (middle)

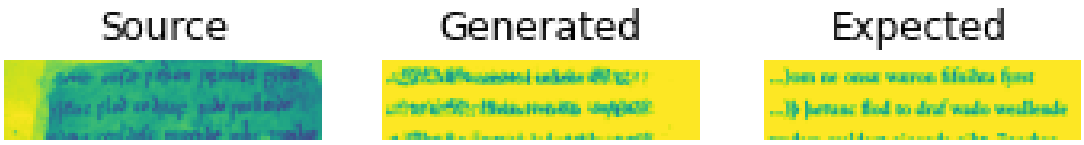


Figure 5.6: The GAN model output on a random image fragment input

5.3 Experimental results for processing whole manuscript pages

The results presented in this section correspond to processing whole manuscript page images. While the original manuscript images and their transcription counterparts are completely aligned, this method suffers from having a very limited number of original images for training the model. The results are presented below.

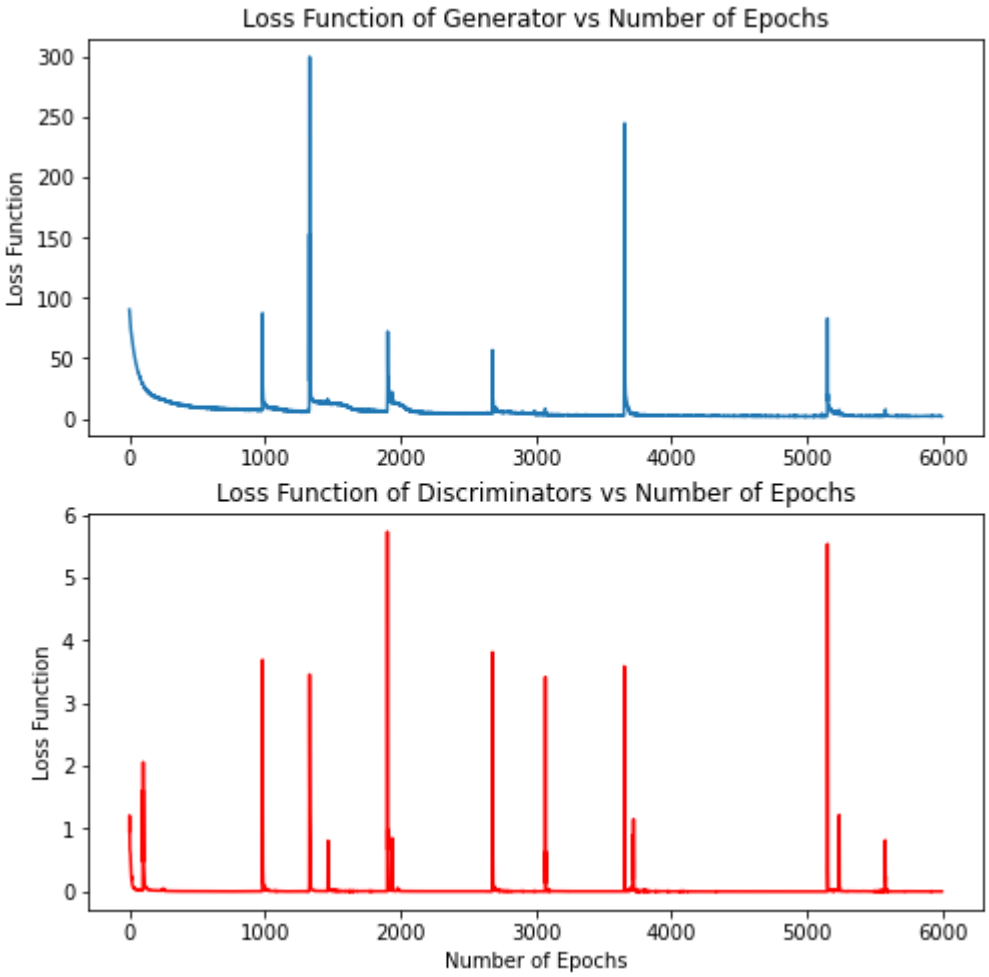


Figure 5.7: The Loss function for training the GAN model on image fragments

Figure 5.7 shows the Loss functions (discriminator and the entire GAN model) values during the training process of 6,000 epochs. The reduced size of the training dataset takes a toll on the training process.

Figure 5.8: Output of GAN model for sample training data: original manuscript image fragment (top), target image (bottom), and model output image (middle)

Figure 5.8 shows a randomly chosen manuscript image page (left), its corresponding transcription snapshots (right), and the generated version of the transcription (middle).

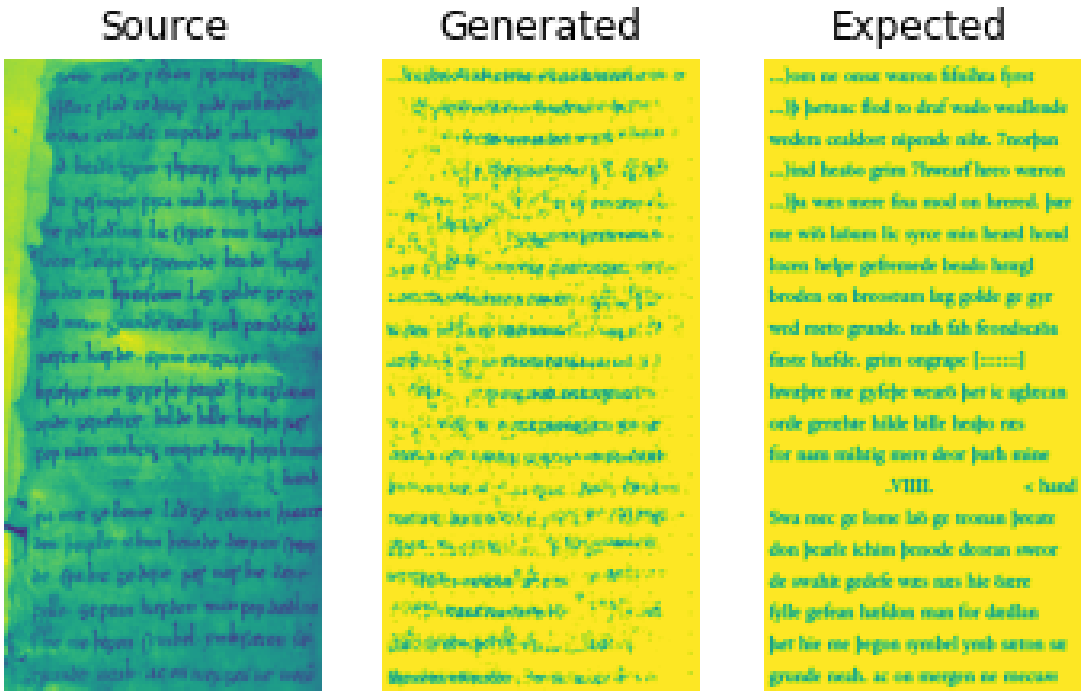


Figure 5.9: The GAN model output on a random image fragment input

Figure 5.9 shows the triplet of original image page (left), original transcription page (right), and the generated transcription image (middle).

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Making historic, rare manuscripts available to the large public has been of a continuous interest for humanities scholars for decades. With the increasing availability of internet and recent technological advances, this has become possible through distribution of static digital images and more complex electronic editions, capable of providing search capabilities and editorial annotations from field experts. However, the recent advances in data processing and artificial intelligence are providing more and more support for automation and better presentation of such historic manuscripts.

In this work we tackle the problem of automatically linking the manuscripts digital images content to the editorial text content (transcription, translations, etc.) created by humanities researches and proposes a framework for a model capable of learning from current published works and produce further knowledge from manuscripts not previously studies. The problem of automatically linking images and text (and subsequently expand search capabilities from text to images) has been of continuing interest for decades. We are making advances in this direction with the model we propose in this work.

Our Generative Adversarial Networks based model takes manuscript digital images and image snapshots of previously created transcriptions and aims to produce new transcription image from new manuscript images. To our knowledge, this is pioneer work in attempting to converting a whole manuscript image into its corresponding transcription image (which in turn, can be easier converted to text). Previous work performed character recognition on character image snippets, manually extracted from manuscript images. Our current work aims to improve those results and produce automatic translation of manuscript images without the tedious work of manually extracting character images. It is an ambitious project and our current results are encouraging.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, Software available from tensorflow.org.

[2] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad, *State-of-the-art in artificial neural network applications: A survey*, Heliyon **4** (2018), no. 11.

[3] Qifang Bi, Katherine E Goodman, Joshua Kaminsky, and Justin Lessler, *What is machine learning? a primer for the epidemiologist*, Oct 2019.

[4] François Chollet et al., *Keras*, `https://keras.io`, 2015.

[5] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath, *Generative adversarial networks: An overview*, CoRR **abs/1710.07035** (2017).

[6] Liang Gonog and Yimin Zhou, *A review: Generative adversarial networks*, 2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA), 2019, pp. 505–510.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, *Generative adversarial nets*, Advances in neural information processing systems **27** (2014).

[8] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye, *A review on generative adversarial networks: Algorithms, theory, and applications*, CoRR **abs/2001.06937** (2020).

[9] Anwarul Mohammad Islam, *Reduced dataset neural network model for manuscript character recognition*, 2020, Master Thesis, Georgia Southern University.

[10] Jayanth Koushik, *Understanding convolutional neural networks*, 2016.

[11] The Editors of Encyclopædia Britannica, *Beowulf old english poem*, https://www.britannica.com/topic/Beowulf/Analysis, 1998.

[12] Keiron O'Shea and Ryan Nash, *An introduction to convolutional neural networks*, 2015.

[13] Adrien Saremi, *Develop a image-to-image translation model to capture local interactions in mechanical networks*, Nov 2020.

[14] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos, *Tensor decomposition for signal processing and machine learning*, IEEE Transactions on Signal Processing **65** (2017), no. 13, 3551–3582.

[15] M. Alex O. Vasilescu, *A multilinear (tensor) algebraic framework for computer graphics, computer vision and machine learning*, 2009.

[16] M. Alex O. Vasilescu and Demetri Terzopoulos, *Multilinear (tensor) image synthesis, analysis, and recognition [exploratory dsp]*, Signal Processing Magazine, IEEE **24** (2007), 118 – 123.

[17] Wikipedia, *Beowulf*, https://en.wikipedia.org/wiki/Beowulf, 2024.

[18] Zhongheng Zhang, *A gentle introduction to artificial neural networks*, Annals of Translational Medicine **4** (2016), no. 19, 370–370.

Appendix A

THE COMPLETE PYTHON CODE FOR EXPERIMENTS

```
# -*- coding: utf-8 -*-
"""
author: Tonilynn Holtz

Our GAN base model v2.5: processes manuscript data
- Black and White
- a square fragment of a whole folio, then sliding right and down
- Generator model enhancement: image -> vector -> image
Based on image2image translation:
    https://github.com/adriensaremi/Springboard/blob/master/Capstone_2%20-%20GAN%20and%20Mechanical%20Networks%20/image2image.ipynb
"""

#%%========================== import libraries
import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt
from IPython import display

#from keras.utils.vis_utils import plot_model
from tensorflow.keras.utils import plot_model

from keras.initializers import RandomNormal
from keras import Model, Input
from keras.models import load_model

from keras.layers import Conv2D, Conv2DTranspose, LeakyReLU, Activation,
    BatchNormalization, Concatenate, Dropout
```

```
from keras.optimizers import Adam

import cv2 as cv #use: pip install opencv-python

print (tf.__version__)


#%%================ functions =========================
def plot_results(images, n_cols=None, unnorm=True):
    '''visualizes fake images'''
    display.clear_output(wait=False)

    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1


    if images.shape[-1] == 1:
        images = np.squeeze(images, axis=-1)

    plt.figure(figsize=(n_cols, n_rows))

    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
        imageorg = image
        if unnorm:
          imageorg = (image +1) * 0.5
        plt.imshow(cv.cvtColor(imageorg, cv.COLOR_BGR2RGB))
        plt.axis("off")

def plot_results2(images, n_cols=None, unnorm=True):
    '''visualizes fake images'''
```

```
    display.clear_output(wait=False)

    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1


    if images.shape[-1] == 1:
        images = np.squeeze(images, axis=-1)

    #plt.figure(figsize=(n_cols, n_rows))
    fig, ax = plt.subplots(1, len(images), figsize=(n_rows, n_cols))
    fig.tight_layout()

    for index, image in enumerate(images):
        #plt.subplot(n_rows, n_cols, index + 1)
        imageorg = image
        if unnorm:
          imageorg = (image +1) * 0.5
        #plt.imshow(cv.cvtColor(imageorg, cv.COLOR_BGR2RGB))
        #plt.axis("off")
        ax[index].imshow(cv.cvtColor(imageorg, cv.COLOR_BGR2RGB))



#%% load, preprocess and plot the training images

DATA_FOLDER = "../mdata2"
TEXT_IMAGES = ['129rt.png', '129vt.png', '130rt.png', '130vt.png',
             '132rt.png', '132vt.png', '133rt.png', '133vt.png',
             '134rt.png', '134vt.png', '135rt.png', '135vt.png',
             '136rt.png', '136vt.png', '137rt.png', '137vt.png',
```

```
90              '138rt.png', '138vt.png', '139rt.png', '139vt.png',
91              '140rt.png', '140vt.png', '141rt.png', '141vt.png',
92              '142rt.png', '142vt.png']
93 MS_IMAGES = ['129r.png', '129v.png', '130r.png', '130v.png',
94              '132r.png', '132v.png', '133r.png', '133v.png',
95              '134r.png', '134v.png', '135r.png', '135v.png',
96              '136r.png', '136v.png', '137r.png', '137v.png',
97              '138r.png', '138v.png', '139r.png', '139v.png',
98              '140r.png', '140v.png', '141r.png', '141v.png',
99              '142r.png', '142v.png']
100
101 IMG_WIDTH = 256
102 IMG_HEIGHT = 512
103 IMG_FRAG_WIDTH = 128
104 IMG_FRAG_HEIGHT = 128
105 IMG_FRAG_SLIDE = 32
106
107 IMG_FRAGnox = (IMG_WIDTH - IMG_FRAG_WIDTH) // IMG_FRAG_SLIDE + 1
108 IMG_FRAGnoy = (IMG_HEIGHT - IMG_FRAG_HEIGHT) // IMG_FRAG_SLIDE + 1
109 IMG_FRAGno = IMG_FRAGnox * IMG_FRAGnoy
110
111 IMG_DIM = (IMG_WIDTH, IMG_HEIGHT)
112 IMG_DIM2 = (IMG_FRAG_WIDTH, IMG_FRAG_HEIGHT)
113
114 WITH_VISUAL_PLOTS = True
115
116 #test image functions
117 if WITH_VISUAL_PLOTS:
118     img = cv.imread(DATA_FOLDER + '/' + MS_IMAGES[0])
119     img_scaled = cv.resize(img, IMG_DIM)
120     gray_image = cv.resize(cv.cvtColor(img, cv.COLOR_BGR2GRAY), IMG_DIM)
```

```
121     #gray_image = cv.cvtColor(img, cv.IMREAD_GRAYSCALE)
122     #thresh = 100
123     bw_image = gray_image #cv.threshold(gray_image, thresh, 255, cv.
    THRESH_BINARY)[1]
124 
125     img = cv.imread(DATA_FOLDER + '/' + TEXT_IMAGES[0])
126     img_scaled = cv.resize(img, IMG_DIM)
127     gray_image = cv.resize(cv.cvtColor(img, cv.COLOR_BGR2GRAY), IMG_DIM)
128     bw_imaget = gray_image
129 
130 
131     fig, ax = plt.subplots(2, IMG_FRAGnox, figsize=(64, 64))
132     fig.tight_layout()
133 
134     delta = IMG_FRAG_SLIDE
135     for i in range(IMG_FRAGnox):
136         imgf = bw_image[0 : IMG_FRAG_HEIGHT,delta*i : delta*i +
    IMG_FRAG_WIDTH]
137         ax[0,i].imshow(cv.cvtColor(imgf, cv.COLOR_BGR2RGB))
138         imgf = bw_imaget[0 : IMG_FRAG_HEIGHT,delta*i : delta*i +
    IMG_FRAG_WIDTH]
139         ax[1,i].imshow(cv.cvtColor(imgf, cv.COLOR_BGR2RGB))
140 
141 
142     plt.show()
143 
144 #%%**************** prepare the training set
    ****************************
145 
146 
147 #(X_train, _), _ = keras.datasets.cifar10.load_data()
```

```
imgs = [None] * (len(TEXT_IMAGES) - 1) * IMG_FRAGno
imgs_ms = [None] * (len(TEXT_IMAGES) - 1) * IMG_FRAGno
delta = IMG_FRAG_SLIDE
ii = 0
for i in range(len(TEXT_IMAGES) - 1):
    img = cv.imread(DATA_FOLDER + '/' + TEXT_IMAGES[i])
    gray_image = cv.resize(cv.cvtColor(img, cv.COLOR_BGR2GRAY), IMG_DIM)
    bw_image = np.reshape(gray_image, (IMG_HEIGHT, IMG_WIDTH, 1)) #cv.threshold(gray_image, thresh, 255, cv.THRESH_BINARY)[1]

    imgms = cv.imread(DATA_FOLDER + '/' + MS_IMAGES[i])
    gray_imagems = cv.resize(cv.cvtColor(imgms, cv.COLOR_BGR2GRAY), IMG_DIM)
    bw_imagems = np.reshape(gray_imagems, (IMG_HEIGHT, IMG_WIDTH, 1))  #cv.threshold(gray_imagems, thresh, 255, cv.THRESH_BINARY)[1]

    for j in range(IMG_FRAGnoy):
        for k in range(IMG_FRAGnox):
            imgs[ii] = bw_image[delta*j : delta*j + IMG_FRAG_HEIGHT, delta*k : delta*k + IMG_FRAG_WIDTH]
            imgs_ms[ii] = bw_imagems[delta*j : delta*j + IMG_FRAG_HEIGHT, delta*k : delta*k + IMG_FRAG_WIDTH]
            ii = ii + 1

X_target = np.array(imgs)
X_source = np.array(imgs_ms)

if WITH_VISUAL_PLOTS:
    plot_results(X_target[0:IMG_FRAGnox], IMG_FRAGnox, False)
    plot_results(X_source[0:IMG_FRAGnox], IMG_FRAGnox, False)

```

```
174 #The GAN Hacks recommend to normalize and scale the pixals to values
       between -1 and 1.
175 # normalize pixel values
176 X_target = X_target.astype(np.float32)
177 X_source = X_source.astype(np.float32)
178
179 #Rescale
180 X_target = (X_target -127.5) / 127.5
181 X_source = (X_source -127.5) / 127.5
182
183 #%%================================== collect test image(s)
       ====================
184 imgs = [None] * IMG_FRAGno
185 imgs_ms = [None] * IMG_FRAGno
186 delta = IMG_FRAG_SLIDE
187 ii = 0
188 i = len(TEXT_IMAGES) - 1
189 img = cv.imread(DATA_FOLDER + '/' + TEXT_IMAGES[i])
190 gray_image = cv.resize(cv.cvtColor(img, cv.COLOR_BGR2GRAY), IMG_DIM)
191 bw_image = np.reshape(gray_image, (IMG_HEIGHT, IMG_WIDTH, 1)) #cv.
       threshold(gray_image, thresh, 255, cv.THRESH_BINARY)[1]
192
193 imgms = cv.imread(DATA_FOLDER + '/' + MS_IMAGES[i])
194 gray_imagems = cv.resize(cv.cvtColor(imgms, cv.COLOR_BGR2GRAY), IMG_DIM)
195 bw_imagems = np.reshape(gray_imagems, (IMG_HEIGHT, IMG_WIDTH, 1))  #cv.
       threshold(gray_imagems, thresh, 255, cv.THRESH_BINARY)[1]
196
197 for j in range(IMG_FRAGnoy):
198     for k in range(IMG_FRAGnox):
199         imgs[ii] = bw_image[delta*j : delta*j + IMG_FRAG_HEIGHT, delta*k
       : delta*k + IMG_FRAG_WIDTH]
```

```
200         imgs_ms[ii] = bw_imagems[delta*j : delta*j + IMG_FRAG_HEIGHT,
    delta*k : delta*k + IMG_FRAG_WIDTH]
201         ii = ii + 1
202
203 XT_target = np.array(imgs)
204 XT_source = np.array(imgs_ms)
205
206 if WITH_VISUAL_PLOTS:
207     plot_results(XT_target[0:IMG_FRAGnox], IMG_FRAGnox, False)
208     plot_results(XT_source[0:IMG_FRAGnox], IMG_FRAGnox, False)
209
210 #The GAN Hacks recommend to normalize and scale the pixals to values
    between -1 and 1.
211 # normalize pixel values
212 XT_target = XT_target.astype(np.float32)
213 XT_source = XT_source.astype(np.float32)
214
215 #Rescale
216 XT_target = (XT_target -127.5) / 127.5
217 XT_source = (XT_source -127.5) / 127.5
218
219
220 #%%The Generator model of the GAN
221 def encoder_block(layer_in, n_filters, batchnorm=True):
222   init = RandomNormal(stddev=0.02)
223   g = Conv2D(n_filters, (3,3), strides =(2,2), padding='same',
     kernel_initializer=init)(layer_in)
224   # Batch normalization and ReLU functions
225   if batchnorm:
226     g = BatchNormalization()(g, training=True)
227   g = LeakyReLU(alpha=0.2)(g)
```

```
228   return g
229
230 def decoder_block(layer_in, skip_in, n_filters, dropout=True):
231   init = RandomNormal(stddev=0.02)
232   g = Conv2DTranspose(n_filters, (3,3), strides=(2,2), padding='same',
       kernel_initializer=init)(layer_in)
233   g = BatchNormalization()(g, training=True)
234   # Add a drop out to the earlier layers of the decoding part of the
       generator.
235   # This is 'cheap way' to regularize the deep neural network.
236   # We drop out half of the input variables from the previous layer.
237   if dropout:
238     g = Dropout(0.5)(g, training=True)
239   # merge with output of corresponding contracting block
240   #g = Concatenate()([g, skip_in])
241   # relu activation
242   g = Activation('relu')(g)
243   return g
244
245 #image_shape=[IMG_FRAG_HEIGHT, IMG_WIDTH, 1]
246 def define_generator(image_shape=(IMG_FRAG_HEIGHT, IMG_FRAG_WIDTH, 1)):
247   init = RandomNormal(stddev=0.02)
248   in_image = Input(shape=image_shape)
249   # encoder model
250   e1 = encoder_block(in_image, 64, batchnorm=False)
251   e2 = encoder_block(e1, 128)
252   e3 = encoder_block(e2, 256)
253   e4 = encoder_block(e3, 512)
254   #e5 = encoder_block(e4, 512)
255   #e6 = encoder_block(e5, 512)
256   #e7 = encoder_block(e6, 512)
```

```
257   # bottleneck, no batch norm and relu
258   b = Conv2D(512, (3,3), strides=(2,2), padding='same',
       kernel_initializer=init)(e4)
259   b = Activation('relu')(b)
260   # decoder model
261   #d1 = decoder_block(b, e7, 512)
262   #d2 = decoder_block(d1, e6, 512)
263   #d3 = decoder_block(d2, e5, 512)
264   d4 = decoder_block(b, e4, 512, dropout=False)
265   d5 = decoder_block(d4, e3, 256, dropout=False)
266   d6 = decoder_block(d5, e2, 128, dropout=False)
267   d7 = decoder_block(d6, e1, 64, dropout=False)
268   # output and model
269   g = Conv2DTranspose(1, (3,3), strides=(2,2), padding='same',
       kernel_initializer=init)(d7)
270   out_image = Activation('tanh')(g)
271   model = Model(in_image, out_image)
272   return model
273
274 generator = define_generator()
275 generator.summary()
276
277 #%%From noise to 32 x 32 x 3
278 if WITH_VISUAL_PLOTS:
279     plot_model(generator, show_shapes=True,
280                show_layer_names=True)
281
282
283
284 #%% generate a batch of noise input (batch size = 16)
285 #test_noise = tf.random.normal([16, noise_input])
```

```
286 img = cv.imread(DATA_FOLDER + '/' + MS_IMAGES[len(MS_IMAGES)-1])
287 gray_image = cv.resize(cv.cvtColor(img, cv.COLOR_BGR2GRAY), IMG_DIM)
288 bw_image = gray_image #cv.threshold(gray_image, thresh, 255, cv.
        THRESH_BINARY)[1]
289 #test_noise = np.reshape(cv.resize(img, IMG_DIM), (1, IMG_HEIGHT,
        IMG_WIDTH, 3))
290 test_noise = (np.reshape(bw_image, (1, IMG_HEIGHT, IMG_WIDTH, 1))
        -127.5) / 127.5
291
292 # feed the batch to the untrained generator
293 delta = IMG_FRAG_SLIDE
294 ii = 0
295 test_image = generator(test_noise[:,delta*ii : delta*ii+IMG_FRAG_HEIGHT,
        delta*ii : delta*ii+IMG_FRAG_WIDTH,:])
296
297 # visualize sample output
298 #plot_results(bw_image, 1)
299 if WITH_VISUAL_PLOTS:
300     plot_results(test_image, 1)
301
302 print(f'shape of the generated batch: {test_image.shape}')
303
304
305 #%%The Discriminator Model to use in the GAN
306
307 def define_discriminator(image_shape):
308   # weight initialization, source and target inputs and merge them
309   init = RandomNormal(stddev=0.02)
310   in_source_image = Input(shape=image_shape)
311   in_target_image = Input(shape=image_shape)
312   merged = Concatenate()([in_source_image, in_target_image])
```

```
313 # CNN: 64 filters
314 hl = Conv2D(64, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(merged)
315 # set alpha to 0.2 rather than the default 0.3 to account for some
     negative pixels
316 hl = LeakyReLU(alpha=0.2)(hl)
317 # CNN: 128 filters
318 hl = Conv2D(128, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
319 # BatchNormalization so that the inputs are standardized (mean = 0,
     std  = 1)
320 hl = BatchNormalization()(hl)
321 hl = LeakyReLU(alpha=0.2)(hl)
322 # CNN: 256 filters
323 hl = Conv2D(256, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
324 hl = BatchNormalization()(hl)
325 hl = LeakyReLU(alpha=0.2)(hl)
326 # CNN: 512 filters
327 hl = Conv2D(512, (3,3), strides=(2,2), padding='same',
     kernel_initializer=init)(hl)
328 hl = BatchNormalization()(hl)
329 hl = LeakyReLU(alpha=0.2)(hl)
330 # 2nd to last output layer
331 hl = Conv2D(512, (3,3), padding='same', kernel_initializer=init)(hl)
332 hl = BatchNormalization()(hl)
333 hl = LeakyReLU(alpha=0.2)(hl)
334 # output layer
335 hl = Conv2D(1, (3,3), padding='same', kernel_initializer=init)(hl)
336 patch_out = Activation('sigmoid')(hl)
337 # define and compile model
```

```
338   model = Model([in_source_image, in_target_image], patch_out)
339   opt = Adam(learning_rate=0.0002, beta_1=0.5)
340   model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights
       =[0.5])
341   return model
342
343 #image_shape = data['arr_0'].shape[1:]
344 #m = define_discriminator(image_shape)
345 # m.summary()
346 image_shape=[IMG_FRAG_HEIGHT, IMG_WIDTH, 1]
347 discriminator = define_discriminator(image_shape)
348
349 discriminator.summary()
350
351 #%% discriminator model
352 plot_model(discriminator, show_shapes=True, show_layer_names=True)
353
354 #%% Creating the GAN with the Generator & Discriminator models.
355 def define_gan(g_model, d_model, image_shape):
356   in_source = Input(shape=image_shape)
357   # discriminator not trainable
358   d_model.trainable = False
359   # generate image from in_source
360   gen_out = g_model(in_source)
361   # pass both in_source and gen_out to discriminator
362   dis_out = d_model([in_source, gen_out])
363   # build a model where image source is compiled against both
       descrimination and generated image
364   model = Model(in_source, [dis_out, gen_out])
365   opt = Adam(learning_rate=0.0002, beta_1=0.5)
366   model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt,
```

```
    loss_weights=[1,100])
  return model


#gan = keras.models.Sequential([generator, discriminator])
#gan.summary()

#%%============================== train function
    ==============================
def generate_real_samples(dataset, n_samples, patch_shape1, patch_shape2
    ):
  trainA, trainB = dataset
  # choose random images from dataset
  ix = np.random.randint(0, trainA.shape[0], n_samples)
  # retrieve selected images
  X1, X2 = trainA[ix], trainB[ix]
  # generate 'real' class labels (1)
  y = np.ones((n_samples, patch_shape1, patch_shape2, 1))
  return [X1,X2], y


def generate_fake_samples(g_model, samples, patch_shape1, patch_shape2):
  X = g_model.predict(samples)
  # create 'fake' class labels (0)
  y = np.zeros((len(X), patch_shape1, patch_shape2, 1))
  return X, y


def summarize_performance(step, g_model, dataset, n_samples=2,
    save_model = True):
```

```
# select a sample of input images and its corresponding fake images
[X_realA, X_realB], _ = generate_real_samples(dataset, n_samples, 1,
 1)
X_fakeB, _ = generate_fake_samples(g_model, X_realA, 1, 1)
# scale all pixels from [-1,1] to [0,1]
X_realA = (X_realA + 1) / 2.0
X_realB = (X_realB + 1) / 2.0
X_fakeB = (X_fakeB + 1) / 2.0
# subplot real source, generated and target images
for i in range(n_samples):
  plt.subplot(3, n_samples, 1 + i)
  plt.axis('off')
  plt.imshow(X_realA[i])
for i in range(n_samples):
  plt.subplot(3, n_samples, 1 + n_samples + i)
  plt.axis('off')
  plt.imshow(X_fakeB[i])
for i in range(n_samples):
  plt.subplot(3, n_samples, 1 + n_samples*2 + i)
  plt.axis('off')
  plt.imshow(X_realB[i])
# save plot to file
filename1 = 'plot_%06d.png' % (step+1)
plt.savefig(filename1)
plt.close()
# save the generator model
if save_model:
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

```
424
425 def train(d_model, g_model, gan_model, dataset, n_epochs=500, n_batch=3)
        :
426   # dataset
427   patch_shape1 = d_model.output_shape[1]
428   patch_shape2 = d_model.output_shape[2]
429   trainA, trainB = dataset
430   # the epochs, the batch per epoch and the total number of training
       iterations
431   bat_per_epo = int(len(trainA) / n_batch)
432   n_steps = bat_per_epo * n_epochs
433   for i in range(n_steps):
434     # select a batch of real samples and update discriminator
435     [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch,
       patch_shape1, patch_shape2)
436     d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
437     # generate a batch of fake samples and update discriminator
438     X_fakeB, y_fake = generate_fake_samples(g_model, X_realA,
      patch_shape1, patch_shape2)
439     d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
440     # update the generator and summarize performance
441     g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
442     print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i+1, d_loss1, d_loss2,
      g_loss))
443         #summarize every 100 epochs and at the last step
444     if ((i+1) == n_steps) or ((i+1) % (bat_per_epo*100) == 0):
445       summarize_performance(i, g_model, dataset)#, save_model=((i+1) ==
      n_steps))
446 #%%============= perform training ================================
447 dataset = [X_source, X_target]
448 print('Loaded', dataset[0].shape, dataset[1].shape)
```

```
449
450 image_shape = dataset[0].shape[1:]
451 # the models
452 d_model = define_discriminator(image_shape)
453 g_model = define_generator(image_shape)
454 gan_model = define_gan(g_model, d_model, image_shape)
455
456 NEPOCHS = 500
457 BATCHS = 2
458 train(d_model, g_model, gan_model, dataset, n_epochs=NEPOCHS, n_batch=
        BATCHS)
459
460 #%% ====================== run some tests from training data
        ====================================
461 # plot source, generated and target images
462 def plot_images(src_img, gen_img, tar_img):
463   images = np.vstack((src_img, gen_img, tar_img))
464   # scale from [-1,1] to [0,1]
465   images = (images + 1) / 2.0
466   titles = ['Source', 'Generated', 'Expected']
467   for i in range(len(images)):
468     plt.subplot(1, 3, 1 + i)
469     plt.axis('off')
470     plt.imshow(images[i])
471     plt.title(titles[i])
472   plt.show()
473
474 # load images and model
475 [X1, X2] = [X_source, X_target]
476 print('Loaded', X1.shape, X2.shape)
477
```

```
478 bat_per_epo = int(len(X_source) / BATCHS)
479 n_steps = bat_per_epo * NEPOCHS
480
481 #filename2 = 'model_%06d.h5' % n_steps
482 filename2 = 'HPC0209/model_101000.h5'
483 model = load_model(filename2)
484
485 # select a random image from the set
486 ix = np.random.randint(0, len(X1), 1)
487 src_image, tar_image = X1[ix], X2[ix]
488 gen_image = model.predict(src_image)
489 plot_images(src_image, gen_image, tar_image)
490
491 plt.imshow(gen_image[0])#.astype('uint8'))
492 plt.show()
493
494 #%%=================== tun test on test data (unseen while training)
    ================================
495 # load images and model
496 [X1, X2] = [XT_source, XT_target]
497 print('Loaded', X1.shape, X2.shape)
498 model = load_model(filename2)
499 #model = load_model('model_000300.h5')
500 #model = load_model('HPC0201/model_023000.h5')
501 model.compile()
502
503 # select a random image from the set
504 ix = [0]# np.random.randint(0, len(X1), 1)
505 src_image, tar_image = X1[ix], X2[ix]
506 gen_image = model.predict(src_image)
507 plot_images(src_image, gen_image, tar_image)
```

```
508
509 plt.imshow(gen_image[0])#.astype('uint8'))
510 plt.show()
511
512 plt.imshow(tar_image[0])#.astype('uint8'))
513 plt.show()
```

Listing A.1: The complete Python code