ASSIGNMENT OF BACHELOR’S THESIS

Title:	Image Inpainting Using Generative Adversarial Networks
Student:	Tomáš Halama
Supervisor:	Ing. Magda Friedjungová
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2020/21

Instructions

Repairing of damaged images has been an important topic in machine learning for a long time. Generative adversarial networks (GANs) have proved to be a suitable tool to repair images and fill the missing part with created “content” (context encoder).

Survey state of the art algorithms for image inpainting focused on convolutional generative adversarial networks.
Implement at least one surveyed algorithm using generative adversarial learning. Compare its performance to different machine learning approaches (e.g., autoencoders) on publicly accessible datasets.
Test your model and experiment with different region mask settings. Examine particular errors and describe the limitations of current algorithms.

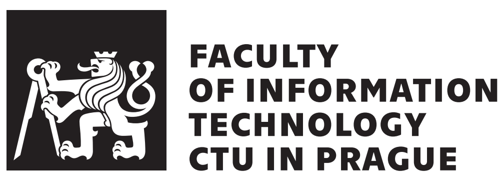
References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 28, 2019



Bachelor's thesis

Image Inpainting Using Generative Adversarial Networks

Tomáš Halama

Department of Applied Mathematics
Supervisor: Ing. Magda Friedjungová

May 13, 2020

Acknowledgements

I would like to express my appreciation for the guidance from the supervisor of this thesis Ing. Magda Friedjungová. She was an exceptional source of support, motivation, and knowledge throughout all phases of this work. I could not have imagined having a better advisor, thank you.

Further thanks go to my partner, Iveta Šárfyová, who shared the daily struggles and efforts with me and managed to make my days brighter than just the computer screen would.

Finally I would like to apologise to my family, closest friends and the Žabla tým for often talking about nothing else than my school-related work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Tomáš Halama. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Halama, Tomáš. *Image Inpainting Using Generative Adversarial Networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

Restoring damaged regions in image data is a relevant and difficult problem, which gets proportionally harder with the severity of the damage. In the last few years we have seen promising progress in tackling this issue using deep learning models. This thesis verifies and compares different approaches to handling the image inpainting problem. Since generative adversarial networks are one of the most inventive and promising architectures, a survey on current methods was performed and two selected methods were reimplemented. Our implementations are compared to other inpainting methods using classification models. The presented results reflect the influence of damage type and damage severity on the ability of each of the considered methods to successfully inpaint a damaged image.

Keywords image inpainting, autoencoder, image corruption, generative adversarial networks, deep learning

Abstrakt

Obnovení poškozených oblastí v obrazových datech je aktuální a náročný problém, jehož obtížnost roste se závažností a velikostí daného poškození. V posledních pár letech lze pozorovat značný pokrok při řešení tohoto problému za pomoci hlubokých neuronových sítí. Tato práce se zabývá ověřením a srovnáním různých přístupů k doplňování obrazových dat. Jelikož generativní adversariální sítě jsou jednou z nejslibnějších architektur, byl v této práci zpracován přehled aktuálních metod a dvě z nich byly reimplementovány pro účely dokreslování. Naše implementace neuronových sítí jsou srovnány s jinými metodami za pomoci klasifikačních modelů. Prezentované výsledky vypovídají o vlivu typu a rozsahu poškození na schopnost jednotlivých metod provést úspěšné dokreslení.

Klíčová slova dokreslování obrazu, autoenkodér, poškození obrazu, generativní adversariální sítě, hluboké učení

Contents

Introduction **1**
Motivation . . . 1
Objectives . . . 2
Structure of the Thesis . . . 2

1 Image Inpainting **3**
1.1 Problem Definition . . . 3
1.2 Generative Adversarial Networks . . . 4
1.3 Denoising Autoencoders . . . 5

2 State-of-the-art **7**

3 Methodology **11**
3.1 Task Definition . . . 11
3.2 Context Encoder . . . 12
3.3 GAIN . . . 14

4 Implementation **19**
4.1 Technologies . . . 19
4.2 Specifics for Context Encoder . . . 20
4.3 Specifics for GAIN . . . 22

5 Experiments **25**
5.1 Datasets . . . 25
5.2 Types of Damage . . . 26
5.3 Damage Severity . . . 27
5.4 Evaluation Method . . . 28

6 Results and Discussion **31**
6.1 Experimental Results . . . 31

6.2 Comparison of Methods 33

Conclusion **37**
Contribution 37
Future Work 38

Bibliography **39**

A Acronyms **43**

B Contents of Enclosed SD card **45**

C Results Tables **47**

D Visual Demonstration of Results **51**

E Network Architectures **61**

List of Figures

3.1 Architecture of a Context Encoder. 13
3.2 Architecture of a GAIN. 15

5.1 An example of the MNIST dataset. 25
5.2 An example of the CIFAR-10 dataset. 26
5.3 Demonstration of square damage. 27
5.4 Demonstration of noise damage within one training dataset. 27

List of Tables

6.1 Classifier accuracy baseline values for undamaged datasets. 32
6.2 Classifier accuracy results for CIFAR-10, centre square damage. . . 32
6.3 Classifier accuracy results for MNIST, corner square damage. . . . 32
6.4 Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged CIFAR-10 dataset. 33
6.5 Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged MNIST dataset. 34

C.1 MSE results for MNIST, centre square damage. 47
C.2 MSE results for MNIST, corner square damage. 47
C.3 MSE results for MNIST, random noise damage. 48
C.4 MSE results for CIFAR-10, centre square damage. 48
C.5 MSE results for CIFAR-10, corner square damage. 48
C.6 MSE results for CIFAR-10, random noise damage. 48
C.7 Classifier accuracy baseline values for undamaged datasets. 48
C.8 Classifier accuracy results for MNIST, centre square damage. . . . 49
C.9 Classifier accuracy results for MNIST, corner square damage. . . . 49
C.10 Classifier accuracy results for MNIST, random noise damage. . . . 49
C.11 Classifier accuracy results for CIFAR-10, centre square damage. . . 49
C.12 Classifier accuracy results for CIFAR-10, corner square damage. . . 50
C.13 Classifier accuracy results for CIFAR-10, random noise damage. . . 50

D.1 Demonstration of the inpainting results for all damage types and methods. Shown on 10% damaged CIFAR-10 dataset. 51
D.2 Demonstration of the inpainting results for all damage types and methods. Shown on 20% damaged CIFAR-10 dataset. 52
D.3 Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged CIFAR-10 dataset. 53
D.4 Demonstration of the inpainting results for all damage types and methods. Shown on 40% damaged CIFAR-10 dataset. 54

D.5 Demonstration of the inpainting results for all damage types and methods. Shown on 50% damaged CIFAR-10 dataset. 55
D.6 Demonstration of the inpainting results for all damage types and methods. Shown on 10% damaged MNIST dataset. 56
D.7 Demonstration of the inpainting results for all damage types and methods. Shown on 20% damaged MNIST dataset. 57
D.8 Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged MNIST dataset. 58
D.9 Demonstration of the inpainting results for all damage types and methods. Shown on 40% damaged MNIST dataset. 59
D.10 Demonstration of the inpainting results for all damage types and methods. Shown on 50% damaged MNIST dataset. 60

E.1 Context Encoder discriminator, 50% centre damaged CIFAR-10. . 62
E.2 Context Encoder autoencoder, 50% centre damaged CIFAR-10, continued in Table E.3. 63
E.3 Context Encoder autoencoder, 50% centre damaged CIFAR-10, continuation of Table E.2. 64
E.4 Context Encoder discriminator, noise damaged CIFAR-10. 65
E.5 Context Encoder autoencoder, noise damaged CIFAR-10, continued in Table E.6. 66
E.6 Context Encoder autoencoder, noise damaged CIFAR-10, continuation of Table E.5. 67
E.7 GAIN generator, CIFAR-10. 68
E.8 GAIN discriminator, CIFAR-10. 68
E.9 GAIN generator, MNIST. 69
E.10 GAIN discriminator, MNIST. 69

Introduction

One of the problems that keeps coming up over and over again ever since the discovery of a camera is that images can become damaged in various ways and it is often essential to repair them. The task of image inpainting requires us to fill in a specified region in an image based on the rest of the picture. Historically, this would be done by a professional artist, who may spend hours or even days restoring a single image or painting. Thankfully, due to recent advances in computer science, there are other methods that can aid us in restoring image data. In addition, we can use inpainting techniques to perform complex image editing, such as the removal of objects, watermarks and even various types of noise.

This thesis is intended for a reader who has a respectable knowledge of the machine learning field. Introductory information about key concepts can be found in other publications [1].

Motivation

Most standard methods used to perform computer-aided inpainting rely on local features such as colours and textures, but they fail to consider the global semantics of the image. These methods work well for cases where image corruption is minor or straightforward to fill in, but not so well for cases with more significant damage, failing to produce reasonable or plausible outcomes [2]. When presented with a solution that respects the semantics of the image, we can for example generate an entire person's face based on an outline of the head. This is not easily done through standard algorithms which are used daily by graphic designers in tools such as Adobe Photoshop. Thus, we can see there might be a desire for a solution that can take on such a feat.

Objectives

A significant number of state-of-the-art methods use deep neural networks and their results look very promising. One of the ways of creating globally well-organised and coherent images is by introducing a second neural network, an adversary, that tries to decide whether the produced results look artificial or genuine. The original generating network can learn to produce results that are much less likely to be discarded as artificial using information from this adversary network. Such networks are called generator and discriminator. This type of architecture called Generative Adversarial Networks (GAN) was proposed in 2014 in [3].

The main aim of this thesis is to:

- survey current state-of-the-art methods, focusing on models that make use of an adversarial discriminator,
- implement two models that use adversarial training,
- experiment with different damage mask settings on various datasets,
- perform a comparison of different approaches.

The performance is demonstrated on different datasets that were damaged in various ways, such as having a missing square region or having pixels dropped based on random noise.

Structure of the Thesis

This work consists of 6 chapters. Chapter 1 aims to thoroughly introduce the image inpainting task and present the outline of two machine learning models, which are a common foundation for many of the state-of-the-art methods presented in the survey in Chapter 2. In Chapter 3, we discuss the needed terminology and notation and follow up with a detailed description of the two models chosen for reimplementation. The technologies used and the steps taken during the design and training of the models are summarised in Chapter 4. As described in Chapter 5, once the models are trained, we perform several types of experiments and evaluate them. These results are then shown and discussed in Chapter 6.

Chapter 1

Image Inpainting

This chapter introduces the image inpainting problem and concepts related to it. We also describe two important types of models that many methods build on in order to tackle the image inpainting task.

1.1 Problem Definition

Restoring image data is a task that has recently been in the spotlight for many researchers all over the world. Many different kinds of images can have areas that we might want to repair or replace. These types of data anomalies include, but are not limited to: blurred areas, watermarks, unwanted objects or even widespread noise. To perform a successful image inpainting, we need to provide a seamless and plausible replacement for a specific region of pixels in the image.

Depending on the usage and meaning, image inpainting is similar to other tasks, such as data imputation or image denoising. The former addresses a more general problem of filling in missing data and the latter aims to reduce noise and improve the sharpness of images. We need to differentiate two basic types of inpainting, blind inpainting and non-blind inpainting. For blind inpainting we do not explicitly say what part of the image needs to be inpainted, the model decides on its own. This can be beneficial for tasks such as removing noise or text from the image, where creating a mask for the designated regions can be complicated. Non-blind inpainting requires the user to specify the area of the image that needs to be changed, which is analogous to how modern graphics editors work when retouching image imperfections.

A lot of methods can be used to solve the task. Most traditional algorithms rely on finding the nearest neighbours and synthesizing textures from the rest of the undamaged data [5]. In order to surpass their results, recent meth-

ods often learn a representation of global context. Working only with local features often ends in results that do not reflect the structure of the original image well. Such an extension of image inpainting to large non-trivial areas is called semantic image inpainting [6].

Even though there are many existing approaches to solving image inpainting, this thesis is mostly focused on more recent state-of-the-art methods with an emphasis on neural networks, such as generative adversarial networks. These kinds of models have a larger learning capacity which enables us to fill in large areas in an image, while preserving a higher level of context. For more information about the other methods, see [5].

1.2 Generative Adversarial Networks

This section introduces and discusses the architecture and principles behind GAN-based models. The main concepts and an outline of the training process is provided.

In machine learning we differentiate between discriminative and generative models. The key difference is that discriminative models seek to learn only the boundary between different classes in the data, while generative models try to capture the actual distribution of the data. Thanks to approximating the training data distribution, we can sample from it and generate new data points, hence the name generative models.

One such type of a generative neural network design was proposed in 2014 by I. Goodfellow [3]. Based on insights from game theory, the training is performed in a way similar to a competition between two networks. The networks are called the generator G and the discriminator D and each has a specific role in the training logic.

Generator G's goal is to be able to output artificial data points, that appear to originate from the same distribution as samples from the original training set. On G's input, there is a vector of random noise variables $\mathbf{z}$, that makes it possible to generate diverse samples. D then tries to differentiate between real samples from training data and data points generated by G. On its output there is a single scalar that represents the probability of the input being real rather than generated.

The networks are trained in a competitive adversarial manner. By producing more plausible outputs, the generator G tries to confuse discriminator D, which in turn learns to improve itself at correctly identifying generated samples. Let $\mathcal{L}_D$ and $\mathcal{L}_G$ denote the loss that D and G try to minimise, re-

spectively. Additionally, $\mathbf{x}$ represents a real sample from training data and $\mathbf{z}$ is the aforementioned random noise vector.

$$\mathcal{L}_D = -\log D(\mathbf{x}) - \log\left(1 - D(G(\mathbf{z}))\right)$$

$$\mathcal{L}_G = \log\left(1 - D(G(\mathbf{z}))\right)$$

Both of these are basically the same loss function, which one network maximises and the other minimises. The only difference is that G has no direct relation to real data $\mathbf{x}$, so the $\log D(\mathbf{x})$ term is left out in the $\mathcal{L}_G$ loss.

Unfortunately, the adversarial architecture is known for its troublesome learning [7]. We need both networks to be well synchronised to ensure learning convergence. Since the only way G learns is through D's opinion, when the discriminator reaches near-perfect results, the gradients start vanishing to a point where it provides no useful information. Another way the model may fail to produce sound results is called mode collapse. It is a scenario where G needs to learn a multi-modal distribution, but instead it maps various realizations of $\mathbf{z}$ to a single output. It should be noted that this particular point can alter between different modes during training.

Even though GAN was originally introduced with multilayer perceptrons in mind, most applications for image data make use of deep convolutional layers [8, 9], as we might see in Chapter 2.

1.3 Denoising Autoencoders

This section briefly introduces the autoencoder architecture and elaborates on its uses for image reconstruction.

An autoencoder is a type of neural network that attempts to replicate its input $\mathbf{x}$ on its output. It consists of two components, an encoder E and a decoder D. The encoder's output is a vector $\mathbf{z} = E(\mathbf{x})$ with lower dimension, also called a latent space vector. There are some uses for a higher dimensional latent space, but we shall neglect this case for the purposes of this thesis. We do not want the network to learn an identity function $D(E(\mathbf{x})) = \mathbf{x}$, but rather make it produce an approximate copy, which still holds the same properties as the original input [10]. As a consequence, the network is encouraged to learn the most important and potentially useful feature representations in the latent space.

The training process minimises loss function $\mathcal{L}$, which measures how different two data points are. Depending on the case, different loss functions can be

used as $\mathcal{L}$. Some of the most popular are mean-squared error and cross-entropy. Respecting the notation of the previous paragraph, the training process aims to minimise the following expression.

$$\mathcal{L}(\mathbf{x}, D(E(\mathbf{x})))$$

Based on the principles behind autoencoders, there was a slight modification presented that enabled us to recover partially damaged data by changing the criteria for reconstruction. Instead of learning to replicate the input directly, the input is damaged beforehand and the network's output is trained to approximate the undamaged original image. In order to create a deep architecture, the authors of [11] stack multiple encoder/decoder pairs, by chaining multiple autoencoders so that each encoder's output is the input to the next. The same thing applies for decoders while respecting the order of encoders. The motivation behind this architecture is to have each latent layer represent more abstract features, similar to how most other deep neural networks work. These types of networks are called Stacked Denoising Autoencoders (SDA) [11]. The process of denoising training together with the usage of deep layers enable the model to build a well-structured and robust hidden layer of features, from which it is possible to reconstruct the original image [10].

Chapter 2

State-of-the-art

The survey performed within this thesis contains the remarks and findings of multiple scientific publications related to the image inpainting task. Two of these methods were chosen to be reimplemented as a part of this thesis.

As one can notice, the denoising process described in Section 1.3 captures the essence of what image inpainting does. Additional research improved the architecture and proposed a method to perform blind image inpainting, as described in Section 1.1. The presented model was named Stacked Sparse Denoising Auto-encoder (SSDA) [12] and achieved significant results, successfully removing substantial noise damage or erasing text from an image foreground.

One of the first works handling semantic inpainting using deep neural networks with an adversary discriminative network was Context Encoders (CE) [2]. Based on the autoencoder architecture and using only convolutional layers, they achieved superior results in a semantic inpainting task when faced against mainstream tools like Adobe Photoshop. The main improvement over previous approaches consists of the usage of an adversarial discriminator loss (as presented in GAN [3]) in addition to the pixel-wise reconstruction loss. Without the adversarial loss the results were not sharp enough, matching only general shapes and colours. The most likely cause was the fact that an average of different inpainting results would have a lower reconstruction loss overall. The discriminator pushes the decoder to produce a specific sample rather than a blurry average. One of the shortcomings of this model is the absence of input noise, which is usually used in most GAN-based models. The consequence of this is that we cannot generate various diverse outputs for one specific input.

Another improvement came with the introduction of the Contextual Attention layer [13]. Building upon previous works, the proposed architecture consists

of two consecutive stages, one for producing a coarse approximate inpainting result and the other one for refining the coarse result. The newly introduced contextual attention layer enables distant areas of the image to influence each other. When combined with two discriminating losses, one for determining whether the entirety of the resulting image is real-looking and one only for the generated patch, the work achieved more plausible results than previously mentioned methods in a human evaluated test.

While reaching impressive results, we are still not able to achieve a balance in combining global context semantics and local textures. Using insights from the Context Encoder architecture and coarse-to-fine CNNs [13] an architecture called Generative Multi-Column Convolutional Neural Network (GMCNN) was presented [14]. The model incorporates two discriminators, one for global features and the other for local features. To be able to extract features at different context levels, there are several parallel columns of encoder/decoder pairs with different convolutional kernel sizes.

Acknowledging that there are many ways to perform inpainting that still look realistic, we might seek a better solution. In an analogy to how two art renovators would each repair a painting in a different way: the general structure might look very similar for both results, but they would differ in fine details. To solve this issue, Pluralistic Image Completion [15] presented a solution that can generate multi-modal results. Building upon Context Encoders, MultiColumn CNN and SAGAN [16] they achieved results with improved consistency while being able to generate diverse samples.

When using standard convolutional layers to extract a feature representation, we take all pixels under the convolutional filter as equal. A problem specific to this task arises when we do not differentiate between valid pixels, generated pixels and invalid pixels during the convolutional computation. This can lead to a colour discrepancy, blurriness or rough edges. Additionally, methods that handle multi-modal outputs lack the ability to be guided by the user towards a specific result [15, 17]. In a practical real-world solution, we might want to have the means to guide the inpainting process using an outline. To address both of these issues, a paper called Free-Form Image Inpainting with Gated Convolution [18] was published. It introduced a method which allows us to specify a rough sketch to which the computation adheres. With the help of a novel architecture for the discriminator (SN-PatchGAN), their method achieves highly competitive results. It should be noted that while the previous models used multiple complicated loss functions, this model uses only two loss measures, pixel-wise reconstruction loss and SN-PatchGAN loss—all this while reaching better results and a considerably shorter training time. This is achieved thanks to the architecture of SN-PatchGAN being able to grasp many previous efforts into a single unit.

A more general problem we previously mentioned is called data imputation. We can use the generative models solving this task to handle image inpainting as well. As published in GAIN [4], the usage of a discriminator network might help us perform a data imputation task in a way that significantly outperforms previous attempts. The main contribution consists of a novel hint mechanism, where we provide the discriminator with additional information. It should be noted that this method only requires us to supply the damaged training data with missing values. We do not need to have a complete undamaged dataset during training, as there is no reconstruction process in place for the missing data. A complete and undamaged dataset might often be unavailable or very hard to obtain. Even though this method was originally not demonstrated on image datasets, additional research suggests it could perform successful image inpainting [19], on par with other methods in the field.

As we can see, there were many innovative ideas and improvements presented in the last few years. For the practical part of this thesis, two papers were chosen for implementation and performance comparison. These selected models are Context Encoder and GAIN, as described in the following chapters. The Context Encoder laid the architectural foundations for many other follow-up papers and GAIN supposedly provides sound results for data imputing, possibly making it suitable for image inpainting as well.

Chapter 3

Methodology

Two different network architectures will be implemented for the practical part of this thesis. Both of these were previously mentioned in Chapter 2. The first one is the Context Encoder [2]. Several variations of the architecture were presented, but we will focus only on GAN-like versions of the model. We will consider two of them, both make use of an adversarial discriminator, but differ in the type of damage done to the input. The other implemented model is GAIN [4]. Although it is not primarily an inpainting technique, it was chosen due to its perceived competitiveness after consulting with the supervisor of this thesis.

3.1 Task Definition

For the purposes of this thesis, we will focus on non-blind restoring of missing pixels. A non-blind inpainting means the model has knowledge of areas that are damaged and are supposed to be inpainted. Given a picture $\mathbf{Z}$ of dimensions $a \times b$, we can represent this knowledge as a boolean matrix $\mathbf{M}$ of the same size, where:

$$m_{i,j} = \begin{cases} 1 & \text{if pixel } z_{i,j} \text{ is valid,} \\ 0 & \text{if pixel } z_{i,j} \text{ is damaged.} \end{cases}$$

As previous chapters might suggest, there are various scenarios for the type of damage done to the data. For evaluating our models we will choose two particular types of damage.

3.1.1 Geometric shapes

The first considered type of damaging images is dropping pixels in a common continuous geometric shape. For practical reasons, we will use a square shape of various sizes removed from different locations. The experiments contain

two such cases. One where a square region is removed from the centre of the image, which allows the model to have some information on all borders of the inpainted area. Also due to the nature of the images in the datasets used, the subject of the image is usually centred, which might make this type of damage more challenging to inpaint. For the second variation, we remove a square area from one of the corners of the image. This leaves the model with the task of inpainting an area where only two surrounding borders are known.

3.1.2 Random noise

Rather than dropping vast areas, this method drops each pixel at random with a given probability. Based on the amount of dropped pixels, inpainting this type of damage might vary greatly in difficulty. For low damage severity, there is lots of information left in the data, so the models can benefit from it in order to create plausible results by using the nearest neighbours of the missing pixels. When the damage is severe enough, the image might only have a fraction of original data left, which requires the model to inpaint the majority of the image, leading us to the need of understanding its semantics once again.

3.2 Context Encoder

The Context Encoder (CE) framework [2] is based on an autoencoder architecture with an adversarial discriminator, while exclusively using convolutional layers with varying kernel sizes and channel counts. Similar to denoising autoencoders, we do not try to reconstruct the input image, but rather generate a patch to fill in the missing or damaged area of the input.

What should be noted is the fact that there were multiple variations of the model architecture presented in the original paper. Some of them did not make use of an adversarial discriminator and thus will be ignored for the purposes of this thesis.

The encoder consists of 2D convolutional operations with a progressively increasing number of channels. Analogously, the decoder is made up of 2D transposed convolutions (sometimes also called deconvolutions). Convolutional operations have the effect of decreasing the width and height of the input while extracting features into deeper channel maps. The transposed convolutions work in an opposite manner, they upscale the image and typically decrease the number of channels. The intermediate bridging layer between these two sections is called the bottleneck layer and is meant to represent the encoded context of the image, hence the name Context Encoder. The size of this layer is dataset dependent and impacts the network's ability to encode the semantics of the input image. The bottleneck layer does not need to be

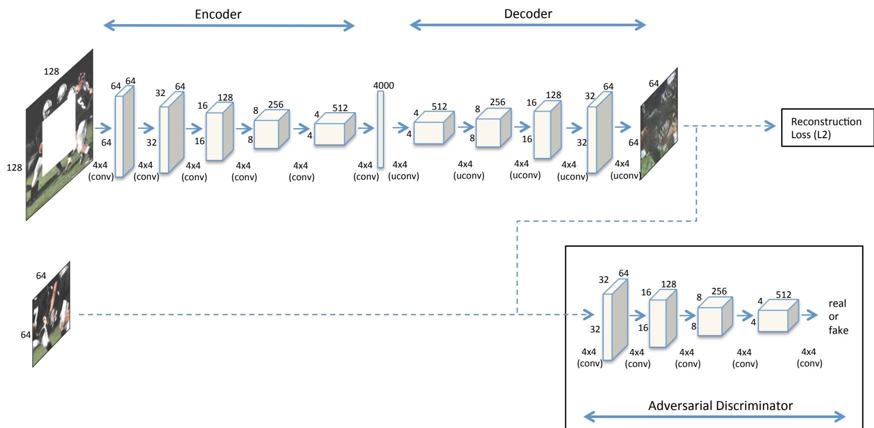


Figure 3.1: Architecture of a Context Encoder [2].

as size-restricted as when training an autoencoder, since we do not have to prevent the learning of the identity function as we do not directly reconstruct the input.

The discriminator has an image input and it is tasked with classifying it either as genuine data or an inpainting result. The network outputs its opinion as a single scalar representing a probability. For arbitrary (or random) region damage, an image of the same size as the original undamaged sample is on the discriminator's input. For the case of square region damage, the authors decided to use only the inpainted patch on the discriminator's input instead of the entire composite image. One of the reasons for this architectural design was the fact that the discriminator could fail to learn useful features and only manage to learn to recognise a boundary of the area where inpainted data were inserted. Overall, patch-only evaluation results in lower computational requirements and in turn a shorter training time.

The entire model is trained using two loss functions. The first one is the reconstruction L_2 loss between the original data and the inpainted result. For the centre square damage, the loss function $\mathcal{L}_{rec}$ can be defined as:

$$\mathcal{L}_{rec} = \|\mathbf{P} - CE(\mathbf{X}')\|_2^2$$

Where $\mathbf{P}$ is the original patch that was removed from the original image $\mathbf{X}$ by damaging it, $\mathbf{X}'$ is the damaged image and CE is the autoencoder. For random noise damage, the loss can be formulated as:

$$\mathcal{L}_{rec} = \|(1 - \mathbf{M}) \odot (\mathbf{X} - CE(\mathbf{X}'))\|_2^2$$

Where $\mathbf{M}$ is the binary mask for the damage, as was described earlier in Section 3.1.

If we stick to using only the reconstruction loss, the results are blurry and very easily discarded as fake. This is likely caused by the network approximating an average patch of all the possibilities for a plausible inpainting result, instead of picking a concrete sample. This issue is alleviated using the aforementioned adversarial discriminator, which learns to identify features that are specific only to generated patches. Through the discriminator's opinion, the CE network is pushed to produce sharper results instead of uncertain averages. The overall architecture including the adversarial discriminator is presented in Figure 3.1.

The adversarial loss $\mathcal{L}_{adv}$ is then formulated as a minimax game, similar to a vanilla GAN. For the square damage scenario, the discriminator D tries to maximise and the CE tries to minimise the following expression:

$$\mathcal{L}_{adv} = \log D(\mathbf{P}) + \log(1 - D(CE(\mathbf{X}')))$$

In the random noise scenario, the loss can be formulated analogously as:

$$\mathcal{L}_{adv} = \log D(\mathbf{X}) + \log(1 - D(CE(\mathbf{X}')))$$

The losses are weighted using two hyperparameters, λ_{adv} for the adversarial discriminator loss and λ_{rec} for the reconstruction loss. The overall loss is then defined as:

$$\mathcal{L} = \lambda_{adv}\mathcal{L}_{adv} + \lambda_{rec}\mathcal{L}_{rec}$$

Similar to vanilla autoencoders, all training is performed in an unsupervised manner, meaning no class label information is supplied to the model and there is no conditioning for the evaluation. As a result, for significantly damaged images the model might perform a successful inpainting, while differing in the class characteristic content.

3.3 GAIN

In contrast to the previous model, a GAIN [4] does not use any convolutional layers, as it is primarily intended for general data, not restricting solely to images. The architecture consists of two networks, a generator G and a discriminator D. Additionally, there is a novel mechanism that is used for generating hints for the discriminator.

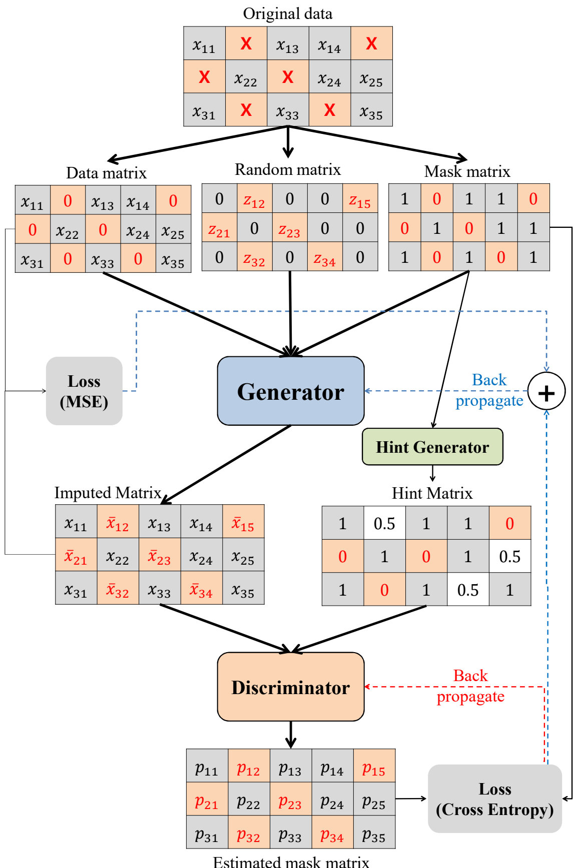


Figure 3.2: Architecture of a GAIN [4].

The generator G has three inputs: the corrupted data $\mathbf{X}$, a binary mask $\mathbf{M}$ (see Section 3.1), and a random noise matrix $\mathbf{Z}$, all of them having dimensions $a \times b$. The generator outputs not only the imputed values, but also a reconstruction of the rest of the input. On its output, there is a single matrix $\mathbf{X}' = G(\mathbf{X}, \mathbf{M}, \mathbf{Z})$, which also has a size of $a \times b$. Outputting an entire reconstructed sample ensures that the network learns to capture the essence of the information contained in the data and also enables simple handling for damage of variable size and shape. The shapes of both G and D are based on autoencoder foundations, as they consist of three layers of fully connected neurons, with the middle one having the lowest amount of learning capacity, fulfilling the function of a bottleneck layer.

The discriminator D functions in a slightly different manner than we are used to from previous models. Firstly, we have to construct the matrices for its input. We take the generator's output $\mathbf{X}'$ and then replace all the values

that were not imputed with the original data to create a composite matrix $\mathbf{C}$, where:

$$\mathbf{C} = \mathbf{M} \odot \mathbf{X} + (1 - \mathbf{M}) \odot \mathbf{X}'$$

As we can see, some values in the $\mathbf{C}$ matrix are genuine and some were imputed. The discriminator's task is to distinguish which components were taken from the matrix $\mathbf{X}'$ and which originate from the input $\mathbf{X}$ from the training set. As a consequence, the output $\mathbf{M}' = D(\mathbf{C}, \mathbf{H})$ of the discriminator is not a single value but rather a matrix with probabilities for each of the components, providing us with an approximation of a binary mask. The discriminator learns to make its approximated mask as close to the actual mask of the damage $\mathbf{M}$ as possible, which effectively makes it able to recognise imputed values.

To aid the process of learning, we supply the discriminator with additional information in the form of a hint matrix $\mathbf{H}$, as shown above. We use a simple generating mechanism to derive the hint from the original mask $\mathbf{M}$ of the damaged components. To create the hint, we preserve most of the values and additionally perform some type of occlusion for a minor subset of the components. The decision on what components are not revealed is made through a random process, such as sampling a probability for each of the components. The values are typically replaced with a constant value of 0.5, the mean of the numerical values of true and false in the binary mask $\mathbf{M}$. In essence, we provide the discriminator with an incomplete mask and we want it to fill in its decision about the few occluded components.

Let $\mathbf{H} \in \{0, 0.5, 1\}^{a,b}$ be the hint matrix of sizes $a \times b$. To identify the components which were occluded when generating the hint, we can construct a matrix $\mathbf{B}$, which functions as a boolean indicator for components which were not revealed to D, where:

$$b_{i,j} = \begin{cases} 1 & \text{if } h_{i,j} = m_{i,j} \\ 0 & \text{if } h_{i,j} = 0.5 \end{cases}$$

Reflecting the formulation presented in [4], there are three loss functions in effect. Two of them are the adversarial loss functions $\mathcal{L}_G$ and $\mathcal{L}_D$ for the generator and the discriminator, respectively. Additionally, the loss function for the generator $\mathcal{L}_{generator}$ composes of $\mathcal{L}_M$ and $\mathcal{L}_G$. Having all the relevant pieces of information available, we can define the following:

$$\mathcal{L}_M = \sum_{j=1}^{b} \sum_{i=1}^{a} m_{i,j} (x_{i,j} - x'_{i,j})^2$$

$$\mathcal{L}_G = -\sum_{j=1}^{b} \sum_{i=1}^{a} (1 - b_{i,j})(1 - m_{i,j}) \log m'_{i,j}$$

$$\mathcal{L}_D = -\sum_{j=1}^{b}\sum_{i=1}^{a}(1-b_{i,j})\left[m_{i,j}\log m'_{i,j} + (1-m_{i,j})\log(1-m'_{i,j})\right]$$

As we can see, the adversarial losses $\mathcal{L}_G$ and $\mathcal{L}_D$ look only at the components occluded by the hint generating mechanism. $\mathcal{L}_M$ is responsible for pushing the generator to output a good reconstruction of the undamaged components observed on its input, meaning it is not computed from any of the imputed values. In an analogy to training an autoencoder, this loss replaces reconstruction loss. To tune the balance between $\mathcal{L}_M$ and $\mathcal{L}_G$ during training, we use hyperparameter α, so that:

$$\mathcal{L}_{generator} = \mathcal{L}_G + \alpha\mathcal{L}_M$$

Chapter 4

Implementation

This chapter contains implementation details. The work makes use of the CIFAR-10 and MNIST datasets, which are described in Section 5.1.

4.1 Technologies

This section summarises the technologies used in the scope of this thesis. We mention all the relevant libraries and platforms used, including the provided computational environments.

We picked Python as the primary programming language for the practical part of this thesis. We made this choice due to its general popularity for machine learning and the fact that its capabilities and library support surpass other languages. The libraries allow us to write readable efficient code and let us focus on the problem itself instead of the implementation details.

For numerical calculations and data transformations, we used the NumPy library [1]. It provides a Python interface to highly optimised algorithms for many types of numerical computations with an emphasis on matrix operations.

Tensorflow 2.0[2], a machine learning framework created by Google, was used to develop, train and test the models. It provides us with an abstract high-level API for defining the network architecture and training the models, as well as a way to perform operations with tensors, analogous to some NumPy features. Tensorflow was created with NumPy in mind and it is easy to transform data between the two, which is beneficial for using them concurrently.

[1] `https://numpy.org/`

[2] `https://www.tensorflow.org/`

The last library that we used extensively was Matplotlib[3]. It is a tool for plotting various types of data in a human readable way. It was used for generating visualizations of the data, monitoring purposes and for subjectively evaluating the results.

As a reference point for the implemented methods, we also perform the experiments on selected standard inpainting methods. We used the implementations available in the OpenCV[4] and scikit-image[5] libraries.

The Jupyter Notebook[6] web application was used for writing the code. While it is not an appropriate choice for using machine learning models in production, the interactivity and visualizations make it a suitable choice for time-efficient prototyping of a model.

Due to high computational requirements, the use of a modern and fast GPU was a necessity for training the models in a manageable time period. Google provides an environment with access to such hardware through their Google Colaboratory[7] platform. Some models built for the purposes of this thesis were trained using this platform, while overcoming its limitations for long-term computations. The rest of them were trained on hardware provided by the supervisor of this thesis.

4.2 Specifics for Context Encoder

This section contains architectural and training details of our implementation of the Context Encoder (CE).

4.2.1 Architecture

There were several variations of the architecture presented in [2]. In our case, we implemented two types of the model. One is used for square region damage and the other is used for random noise damage. The key difference is that for square region damage, the autoencoder outputs only the patch which was cut from the original picture. For the noise damage, the autoencoder's output is the same size as the damaged sample on the input. The discriminator of each damage type holds the dimension of its input equal to the respective autoencoder's output. The implemented square damage architecture variant for 50% damaged CIFAR-10 is thoroughly described in Tables E.1 to E.3. Analogously, the architecture variant for the random noise damage scenario for

[3]https://matplotlib.org/
[4]https://opencv.org/
[5]https://scikit-image.org/
[6]https://jupyter.org/
[7]https://colab.research.google.com

the same dataset is presented in Tables E.4 to E.6. The architectures for the other severities and datasets are almost the same except for minor differences related to the size of the input and the output.

The encoder consists of multiple blocks, where each block contains a convolutional layer, batch normalization, a leaky rectified unit (ReLU) activation function and possibly a 2D max-pooling layer to reduce the size of processed data. Due to the size of the network's output being dependent on the extent of the damage, the architecture varies slightly for each percentage of damaged data. The differences are minor, such as different kernel sizes or strides for some of the layers. Tables E.1 to E.3 assume a square covering 50% of the input missing, which is the most severe damage considered.

The decoder holds a similar block structure to the encoder with different key components. A single block consists of a transposed convolution, batch normalisation, and a ReLU activation. To mirror the max-pooling operation from the encoder, we use strides on some of the transposed convolutional layers, which leads to an increase of width and height.

The last architectural component that remains is the discriminator. By following the guidelines mentioned by the authors [2, 8], we used several blocks with strided convolutions, batch normalisation layers, Leaky ReLU activation functions and dropout layers. After the last block, there is a sigmoid-activated dense layer with one output, which represents the discriminator's opinion.

One of the mechanisms not implemented in this thesis is a channel-wise dense layer in the bottleneck part of the autoencoder. The dimensions of images in our dataset were smaller than in the original implementation, so a regular dense layer was manageable from a computational perspective.

Another feature we left out from our implementation to simplify it is that for the case when a central square region was missing, the original model performed a reconstruction of a slightly larger area than was actually needed. We suppose this would help the model perform a better transition at the border of the inpainted area and the original image.

4.2.2 Training and Hyperparameter Tuning

The training updates weights of both the autoencoder and the discriminator in each iteration. Since it might be hard to estimate the number of epochs for training, we followed a value presented in [2]. The authors mention 100,000 iterations to be sufficient for training the networks in their case.

For the square damage scenario, our damage sizes vary from 10% to 50% with a different model architecture for each of them. After initial experiments, we decided to use a variable amount of epochs that progressively increases with the amount of damage, because small damage severities often converged much faster than larger severities. Since we use a batch size of 128 and datasets with tens of thousands of training samples, we settled on using 700 epochs for 10% of damage. We added 200 training epochs, for each additional 10% of damage, leading to the following formula:

$$\text{EPOCHS} = 500 + 2000 \cdot \text{DAMAGE},$$

where EPOCHS is the number of epochs and DAMAGE is the percentage of damaged data. For training using the noise damaged dataset, there was a single model for all missing data severities. This is thanks to the architecture staying the same, due to the model always outputting the entire image. We used 1500 epochs, which is the same amount of epochs as for the largest considered amount of square damage.

The losses were implemented as described in Section 3.2, using cross-entropy and L_2 loss functions provided in the Tensorflow and Keras libraries. The losses' weights were suggested to be $\lambda_{adv} = 0.001$ and $\lambda_{rec} = 0.999$ according to the original paper. We found that these hyperparameter values give us sound results for the CIFAR-10 dataset, which is similar to the datasets used in the original paper. On the other hand, the simpler MNIST dataset inpainting results were significantly blurred, which led us to settle with changing the ratio of weights to $\lambda_{adv} = 0.1$ and $\lambda_{rec} = 0.9$. Using these updated weights for CIFAR-10 resulted in significant smudges, which distorted the shapes of inpainted objects.

The original paper mentions the use of a SGD optimiser. For the MNIST dataset, we found that the usage of SGD led to the model often failing to learn any useful features and getting stuck in a local minimum by producing a solid black color for all output pixels. This was alleviated by changing the optimiser to Adam, with an empirically chosen learning rate value of 0.0001. To remain consistent, the same optimiser was used for the CIFAR-10 dataset.

4.3 Specifics for GAIN

This section contains architectural and training details of our implementation of the GAIN.

4.3.1 Architecture

Both the generator and discriminator share a very similar structure. They both consist of three fully-connected layers with a hyperbolic tangent activa-

tion function, where the middle layer has the lowest number of parameters. Authors propose the number of neurons in the first and third layer to be equal to the number of values in a single sample in the dataset. For the middle layer the parameter count is halved.

We followed this methodology for deciding the number of parameters for the MNIST dataset implementation, but we performed a slight modification for the CIFAR-10 dataset. By default, the model's damage mask considers each single value separately, so the discriminator makes a decision about the origin of each of the components on the input. Since CIFAR-10 contains RGB pixels, we do not need the discriminator to decide for each colour channel separately, so we use a single channel mask throughout the entire model. This results in neuron counts divided by a factor of three in some layers, which can be seen in detail in Tables E.7 to E.10.

The hint generating mechanism produces a hint of the same size as the damage mask. As mentioned in the previous paragraph, we use a single channel damage mask for both datasets, which makes the hint a 2D matrix. The hint is generated by occluding some components of the mask. After consulting with the supervisor of this thesis, we implemented the occlusion of each component as a random event with a 10% probability.

4.3.2 Training and Hyperparameter Tuning

As opposed to the Context Encoder, training the GAIN to inpaint a centre square damaged dataset might prove troublesome. In our case, a centre damaged dataset has the same damage mask for all the samples, which makes the discriminator's task trivial. Similar observation can be made for corner square damage. As a consequence, after consulting with the supervisor, it was decided to train only a single GAIN model per dataset, using the noise damaged dataset which contained all the damage severities considered (see Section 5.3).

To decide the number of epochs, we used a validation set and calculated MSE at the end of each epoch. When the error stopped improving, an early stopping was performed. We found that this method of deciding the epoch count almost never leads to more than 100 epochs. Such a relatively short training allowed us to experiment with different hyperparameters. By performing a validation grid search using several learning rate values, optimisers (Adam, RMSprop, SGD) and α values, we ended up using Adam with a learning rate of 0.00001 and $\alpha = 30$ for the CIFAR-10 dataset and SGD with a learning rate of 0.00001 and $\alpha = 13$ for the MNIST dataset. These combinations of values achieved the lowest MSE scores.

Chapter 5

Experiments

This chapter describes the experiments performed in this work. Their design and evaluation process are presented and discussed.

5.1 Datasets

To implement and evaluate the models, two particular datasets were selected. Both of them were chosen due to being well-known and easy to acquire and work with.

The first one is the MNIST dataset [20]. It consists of 70,000 hand-written digits, from which we set aside 21,000 as testing data. A single sample's size is 28×28 pixels and since all the images are black-and-white, it contains only one colour channel. Each sample is labelled based on the digit which is portrayed in the image. Overall there are 10 classes representing each of the digits 0–9.

The second dataset we used was the CIFAR-10 [21] collection. It contains 60,000 samples of which 18,000 were used as testing data. The dataset is made up of low-resolution images of real-life objects, such as various types of

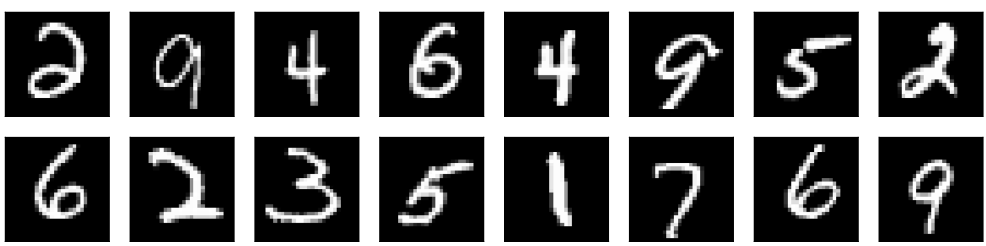


Figure 5.1: An example of the MNIST dataset.

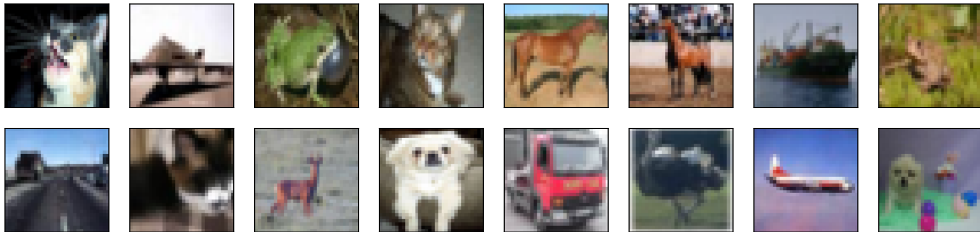


Figure 5.2: An example of the CIFAR-10 dataset.

animals and vehicles. Each image's size is 32×32 pixels, with RGB colour channels. The dataset, again, consists of 10 classes, such as horses, dogs, airplanes or ships.

Both datasets were made publicly available. The Keras [22] library provides both of these datasets through the `keras.datasets` module and since it is bundled with the TensorFlow framework, it served as a source of the data for the implementation presented in this thesis.

Going with the thesis supervisor's advice, training and testing datasets have a 70:30 ratio for all methods. For cases where a validation set is used, it is created by further splitting the training set using a ratio of 75:25.

5.2 Types of Damage

All the implemented methods were evaluated on multiple types of image damage. We prepared several derived datasets for each of the datasets described in Section 5.1 by performing one of the three considered types of damage.

The first two damage types result in a square region missing from the image. These types of damage differ in the location of the missing square. One has the central region dropped (see Figure 5.3a), meaning that the damage mask is equal for all the samples in the dataset. The other has one of its corners missing (see Figure 5.3b), with the specific damage mask periodically alternating four corners. These two types of damage aim to test semantic inpainting, providing the model with either four surrounding borders for central square damage or two borders for the corner square scenario.

The final damage type considered is noise damage. It is performed by dropping each pixel at random with a given probability. This probability is given by the severity of the damage.

(a) Centre square damage. (b) Corner square damage.

Figure 5.3: Demonstration of square damage.
Left to right: original image, cut area, damaged image showing red pixels instead of the removed areas.

5.3 Damage Severity

For each of the damage types described above, we evaluated each model and method for different damage severities. We considered 10%, 20%, 30%, 40%, and 50% of damage done to the images.

Figure 5.4: Demonstration of noise damage within one training dataset.
Left to right: original image, damage mask, damaged image shows red pixels instead of the removed pixels. Top to bottom: 10%, 20%, 30%, 40%, and 50% damage.

Due to the architectural changes needed for one of the methods for each damage severity (see Section 4.2.1), we had to keep each percentage of both square damage types in a separate dataset. This fact lead us to training 10 variations of the Context Encoder model for one dataset: 5 central damage and 5 corner damaged variations. For the noise damage, we kept all the severities within a

single dataset, while ensuring that all the severities are uniformly distributed, as shown in Figure 5.4. To summarise, for each of the two used datasets, we trained 11 Context Encoder models and 1 GAIN model (noise training only).

5.4 Evaluation Method

We picked two methods to evaluate the results of the experiments. Both of them were calculated for each combination of dataset, damage type and damage severity. For comparison, we also included two inpainting methods that do not use neural networks.

The first metric is calculated as a mean squared error (MSE) between the original image $\mathbf{X}$ from the test dataset and the corresponding inpainting result $\mathbf{X}'$. Assuming both images have the same dimensions of $a \times b$ and one colour channel, the MSE metric can then be formulated as:

$$MSE(\mathbf{X}, \mathbf{X}') = \left(\frac{1}{a \cdot b}\right) \sum_{i=1}^{a} \sum_{j=1}^{b} (x_{i,j} - x'_{i,j})^2$$

The metric can be analogously extended to more colour channels, using the channels as another dimension of the image.

The limitation of directly comparing pixel values is that our model often has to make up its own shapes and colours, because there is not enough information to decide confidently. If the model was trained to reduce only the mean squared error, the results would likely be significantly blurred, as mentioned in Section 3.2. In our case the mean squared error is the most meaningful for evaluating noise damage inpainting, where the damaged areas are not continuous and lots of features are kept throughout the image. The remaining features are rather dense, which does not leave a lot of room for novel structures introduced by the inpainting method.

To address the issues of the previous metric, the second evaluation tests the model's ability to recover and preserve the semantic information in the damaged image. It is calculated as an accuracy metric of a classifier trained on an undamaged training set and tuned on an undamaged validation set.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of predicted samples}}$$

The classifier models are based on the code used in examples on the Keras[8] and Tensorflow[9] websites. The architecture consists of 3 convolutional layers

[8] https://keras.io/examples/cifar10_cnn/
[9] https://www.tensorflow.org/tutorials/images/cnn

and 2 dense layers for the MNIST dataset and 4 convolutional layers with 2 dense layers for the CIFAR-10 dataset. Both were trained using a RMSprop optimiser and an early stopping mechanism which monitors the accuracy on the validation set in order to prevent overfitting.

We evaluate all the inpainting methods on each of the test dataset variations damaged in various ways for both of the mentioned metrics. Altogether, we evaluated 2 methods implemented in this thesis and 2 standard methods for comparison. The first standard method is a solution based on the biharmonic equation [23], which is a differential equation that originated from researching linear elasticity in physics. We used the implementation available in the scikit-image[10] library. The second reference inpainting method comes from Alexandru Telea and is based on a fast marching method as presented in [24]. Its implementation was taken from the OpenCV[11] library.

[10] https://scikit-image.org/
[11] https://opencv.org/

Chapter 6

Results and Discussion

This chapter presents the results achieved by our implementation in comparison with two other standard methods. In order to discover and state the limitations of each method, both the empirical and numerical results are discussed.

6.1 Experimental Results

All the numerical results are presented in the form of a table in Appendix C. The MSE results for MNIST are presented in Tables C.1 to C.3 and the CIFAR-10 results are in Tables C.3 to C.5. The classification results include baseline values, which were measured on undamaged original data, as can be seen in Table C.7. The classification accuracy results for inpainted images are shown in Tables C.8 to C.10 for MNIST and in Tables C.11 to C.13 for CIFAR-10.

As a visual comparison for the various types of damage and results of both our and standard methods, we included an exhaustive set of tables for both datasets in Appendix D. Please note that these might not be entirely representative, since the demonstration is limited to two testing samples per dataset.

We selected few of the aforementioned tables for inclusion in the main text. Two of the tables for empirical visual comparison were included in this chapter as a demonstration of the results. The results are shown for 30% damaged CIFAR-10 and MNIST datasets in Tables 6.4 and 6.5. For some of the demonstrated visual results, we also included the numerical classification accuracy results, namely for centre damaged CIFAR-10 in Table 6.2, corner damaged MNIST Table 6.3 and a table with baseline values measured on undamaged data in Table 6.1.

Table 6.1: Classifier accuracy baseline values for undamaged datasets.

Dataset	Train	Validation	Test
MNIST	0.9970	0.9848	0.9866
CIFAR-10	0.6792	0.6912	0.6956

The classification accuracy dropped with the severity of damage as expected. Since the baseline accuracy was lower for the CIFAR-10 dataset to begin with, even the best results often struggled to stay above 0.4 accuracy for this dataset. For the MNIST dataset the measured accuracies also dropped significantly with damage, but the best methods never went lower than 0.6. In general, we could see that the noise damaged datasets achieved the highest accuracy scores. This is expected since this type of damage could likely be plausibly inpainted using significantly simpler methods, such as averaging the nearest neighbours, as there is a lot of information left throughout all regions of the image. The second best accuracy scores were obtained on corner damaged datasets. In this case, a lot of information was still left for the classifier, since the images often have their subjects centred. The lowest accuracy scores were achieved on the centre square damaged datasets. Here the methods were tasked to provide us with the most crucial sections of the image, which tests their ability to uncover and work with the semantics of the image.

Table 6.2: Classifier accuracy results for CIFAR-10, centre square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.6392	0.5794	0.6519	0.6455
20%	0.5880	0.4915	0.5936	0.5809
30%	0.5467	0.4378	0.5467	0.5314
40%	0.4329	0.2739	0.3963	0.3690
50%	0.3660	0.2012	0.3207	0.2882

Table 6.3: Classifier accuracy results for MNIST, corner square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.9866	0.9858	0.9856	0.9853
20%	0.9761	0.9596	0.9478	0.9370
30%	0.9631	0.9125	0.8845	0.8589
40%	0.9276	0.7696	0.7518	0.7371
50%	0.8520	0.5902	0.5584	0.5590

Table 6.4: Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged CIFAR-10 dataset.

	(a) Original	(b) Damage mask	(c) CE	(d) GAIN	(e) Biharmonic	(f) Telea
Centre						
Corner						
Noise						

6.2 Comparison of Methods

As mentioned in Section 5.4, the GAIN and Context Encoder (CE) are faced against two standard inpainting methods, which we call biharmonic and Telea.

We anticipated the standard methods to perform well for inpainting smaller patches and damage severities, but to not be sufficient for larger damages, where a deeper understanding of the input would be more beneficial.

The Context Encoder performs well for both metrics in both square-damaged scenarios. Especially for the classifier metric on centre square missing, it beats all the other methods in the majority of tested damage severities and datasets, while still suffering a decrease in accuracy of circa 0.3 in comparison to undamaged dataset. The centre square missing scenario is arguably the most demanding test case, since the subjects in the images are often centred. Context Encoder has often proven to be able to supply enough information

Table 6.5: Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged MNIST dataset.

to correctly classify even a vastly damaged input. It should be noted that especially for the MNIST dataset, the Context Encoder sometimes manages to perform a convincing inpainting, but due to the severity of the damage it changes the class the image belongs to, leading to an incorrect classification. It is up to the specific potential usage of the model whether this is a successful result or a misstep. These impressive results for square damage are also reflected in the MSE metric, where it achieves similar qualities.

For the square damage scenarios on the MNIST dataset, the GAIN model results are not as sharp and plausible as the Context Encoder. This fact negatively reflected on the classification accuracy, although it still approximately matched the standard methods. As mentioned in Section 4.3.2, the GAIN model was trained exclusively on noise damaged datasets, which is the likely cause. GAIN was also trained with an early stopping mechanism monitoring MSE on the validation set, which means it does not aim for the most real

looking result, but rather an average approximation with the lowest error. A similar phenomenon is mentioned in the description of the Context Encoder in Section 3.2, where it leads to more uncertainty and blurriness in the result. For the square damaged CIFAR-10 dataset GAIN fails to produce results with a familiar structure and its inpainted regions are often reminiscent of random noise, which made it one of the worst methods in terms of MSE and accuracy.

It should be noted that while GAIN has not managed to leverage the context provided in its input, it was also never shown the true regions before damaging. GAIN, as an imputation method, learns on damaged data only. This, together with the fact that it was never trained on square damaged regions, puts it in a significant disadvantage in comparison to the Context Encoder, which had a different model and training for each damage severity on square-damaged datasets and one separate model dedicated to noise damage. This limitation of GAIN has proved to be beneficial for the noise damage scenario, as it achieves the lowest mean squared error of all methods on the CIFAR-10 dataset in all the considered severities. This quality was unfortunately not reflected in the classifier accuracy, where it performed the same or worse than other methods.

One obvious advantage that our implemented models had was the fact that the standard methods had no training and thus no inherent knowledge about the data they were evaluated on. Respecting this, the standard methods performed surprisingly well on the noise damage scenario, even surpassing our implemented methods in some of the experimental measurements. The biharmonic method achieved superior results in the classifier metric for the noise-damaged CIFAR-10 by beating all the other methods (including Context Encoder). After a subjective examination, the biharmonic method provides a very smooth transition for the noise damage mask even in significant severities. This shows the biharmonic method is able to provide very good approximations over small patches of damage, while acting as a spilling blur operation for larger patches.

Conclusion

Image inpainting is a task that benefits greatly from modern neural network architectures. In this thesis, we surveyed several state-of-the-art approaches and implemented two of them. The requirement to understand the semantics of the image became apparent after experimenting with the methods. This chapter aims to summarise our contribution and to outline possible future work.

Contribution

This section looks back at the objectives outlined at the beginning of this text. All the key points of the survey, implementations, experiments and evaluations were fulfilled and should stand as a basis for additional work.

The first part of this thesis presented the image inpainting task and laid basic theoretical foundations needed throughout the text. We performed a survey on machine learning methods dealing with image inpainting and summarised their key concepts and findings into several paragraphs.

Based on the survey, we picked two models that utilised an adversarial discriminator. The choice was made to focus on Context Encoder [2] and GAIN [4]. We implemented these two models using Python and trained them in accordance to the papers they were presented in. The implementations were tuned with respect to both of the training datasets.

While we did manage to achieve results that were acceptable in most of the expected cases, there is a lot of room for improvement and additional testing. The original paper for the Context Encoder [2] used datasets and architectures that were more complex. The simplifications of both of these aspects and minor training differences could cause the decline in quality. The GAIN

model is a general data imputing method and we tested its performance for inpainting various types of damage.

For the comparison we picked two other standard methods and performed an evaluation of MSE between the inpainted and original regions. We also trained a classifier on undamaged data and measured how much the accuracy of predictions differed between undamaged and inpainted test data. Each of these metrics targets a different quality of the inpainting model, as a low average error is not a necessary indication of an acceptable result.

The results have shown that the Context Encoder manages to provide the classifier with enough semantic information to aid the classification significantly. On the other hand, the GAIN model failed to provide reasonable results for the square damage scenario, while reaching low reconstruction error rates for noise damaged data. Interestingly enough, although the noise damaged datasets were simpler to inpaint, in some cases our implementations did not perform as well as the standard methods.

Future Work

Given that the training procedures were different for the two implemented models, it might be insightful to further analyse the influence of training a model on multiple damage severities at once or separately. As an extension, we propose more complex damage scenarios, since square-region damage and random noise damage are both extreme cases, which likely would not happen in the real world.

The Context Encoder provided a basic architectural design, which inspired many other works. These more recent types of models are more elaborate and enable us to do feats such as high-resolution inpainting (see Chapter 2) or even video inpainting [25]. The results presented in this thesis might provide a baseline for additional works concerning this topic.

Since we believe that the GAIN architecture is capable of achieving better results, it might be worth further investigating its training process and the effect of minor architectural and methodological changes, such as the hint generation logic and the inclusion of different damage types and severities in one training process.

Bibliography

1. MURPHY, Kevin P. *Machine learning: a probabilistic perspective*. MIT press, 2012. ISBN 0262018020.
2. PATHAK, Deepak; KRAHENBUHL, Philipp; DONAHUE, Jeff; DARRELL, Trevor; EFROS, Alexei A. Context Encoders: Feature Learning by Inpainting. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016. Available from DOI: 10.1109/cvpr.2016.278.
3. GOODFELLOW, Ian; POUGET-ABADIE, Jean; MIRZA, Mehdi; XU, Bing; WARDE-FARLEY, David; OZAIR, Sherjil; COURVILLE, Aaron; BENGIO, Yoshua. Generative Adversarial Nets. In: GHAHRAMANI, Z.; WELLING, M.; CORTES, C.; LAWRENCE, N. D.; WEINBERGER, K. Q. (eds.). *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 2672–2680. Available also from: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.
4. YOON, Jinsung; JORDON, James; SCHAAR, Mihaela van der. GAIN: Missing Data Imputation using Generative Adversarial Nets. In: DY, Jennifer; KRAUSE, Andreas (eds.). *Proceedings of the 35th International Conference on Machine Learning*. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, vol. 80, pp. 5689–5698. Proceedings of Machine Learning Research. Available also from: http://proceedings.mlr.press/v80/yoon18a.html.
5. BERTALMÍO, Marcelo; CASELLES, Vicent; MASNOU, Simon; SAPIRO, Guillermo. Inpainting. In: *Computer Vision: A Reference Guide*. Ed. by IKEUCHI, Katsushi. Boston, MA: Springer US, 2014, pp. 401–416. ISBN 978-0-387-31439-6. Available from DOI: 10.1007/978-0-387-31439-6_249.

6. YEH, R. A.; CHEN, C.; LIM, T. Y.; SCHWING, A. G.; HASEGAWA-JOHNSON, M.; DO, M. N. Semantic Image Inpainting with Deep Generative Models. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6882–6890. ISSN 1063-6919. Available from DOI: `10.1109/CVPR.2017.728`.
7. SALIMANS, Tim; GOODFELLOW, Ian; ZAREMBA, Wojciech; CHEUNG, Vicki; RADFORD, Alec; CHEN, Xi. Improved Techniques for Training GANs. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Barcelona, Spain: Curran Associates Inc., 2016, pp. 2234–2242. NIPS'16. ISBN 9781510838819.
8. RADFORD, Alec; METZ, Luke; CHINTALA, Soumith. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In: BENGIO, Yoshua; LECUN, Yann (eds.). *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. Available also from: `http://arxiv.org/abs/1511.06434`.
9. GOODFELLOW, Ian J. NIPS 2016 Tutorial: Generative Adversarial Networks. *CoRR*. 2017, vol. abs/1701.00160. Available from arXiv: `1701.00160`.
10. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. ISBN 0262035618. Available also from: `http://www.deeplearningbook.org`.
11. VINCENT, Pascal; LAROCHELLE, Hugo; BENGIO, Y.; MANZAGOL, Pierre-Antoine. Extracting and composing robust features with denoising autoencoders. In: 2008, pp. 1096–1103. Available from DOI: `10.1145/1390156.1390294`.
12. XIE, Junyuan; XU, Linli; CHEN, Enhong. Image Denoising and Inpainting with Deep Neural Networks. In: PEREIRA, F.; BURGES, C. J. C.; BOTTOU, L.; WEINBERGER, K. Q. (eds.). *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 341–349. Available also from: `http://papers.nips.cc/paper/4686-image-denoising-and-inpainting-with-deep-neural-networks.pdf`.
13. YU, Jiahui; LIN, Zhe; YANG, Jimei; SHEN, Xiaohui; LU, Xin; HUANG, Thomas S. Generative Image Inpainting with Contextual Attention. *CoRR*. 2018, vol. abs/1801.07892. Available from arXiv: `1801.07892`.
14. WANG, Yi; TAO, Xin; QI, Xiaojuan; SHEN, Xiaoyong; JIA, Jiaya. Image Inpainting via Generative Multi-column Convolutional Neural Networks. In: BENGIO, S.; WALLACH, H.; LAROCHELLE, H.; GRAUMAN, K.; CESA-BIANCHI, N.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 331–340. Available also from: `http://papers.nips.cc/paper/7316-`

image-inpainting-via-generative-multi-column-convolutional-neural-networks.pdf.

15. ZHENG, C.; CHAM, T.; CAI, J. Pluralistic Image Completion. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 1438–1447. ISBN 978-1-7281-3293-8. ISSN 2575-7075. Available from DOI: 10.1109/CVPR.2019.00153.

16. ZHANG, Han; GOODFELLOW, Ian; METAXAS, Dimitris; ODENA, Augustus. Self-Attention Generative Adversarial Networks. In: CHAUDHURI, Kamalika; SALAKHUTDINOV, Ruslan (eds.). *Proceedings of the 36th International Conference on Machine Learning*. Long Beach, California, USA: PMLR, 2019, vol. 97, pp. 7354–7363. Proceedings of Machine Learning Research. Available also from: http://proceedings.mlr.press/v97/zhang19d.html.

17. BAO, J.; CHEN, D.; WEN, F.; LI, H.; HUA, G. CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2764–2773. ISBN 978-1-5386-1032-9. ISSN 2380-7504. Available from DOI: 10.1109/ICCV.2017.299.

18. YU, J.; LIN, Z.; YANG, J.; SHEN, X.; LU, X.; HUANG, T. Free-Form Image Inpainting With Gated Convolution. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 4470–4479. ISBN 978-1-7281-4803-8. ISSN 2380-7504. Available from DOI: 10.1109/ICCV.2019.00457.

19. LI, Steven Cheng-Xian; JIANG, Bo; MARLIN, Benjamin M. MisGAN: Learning from Incomplete Data with Generative Adversarial Networks. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. Available also from: https://openreview.net/forum?id=S1lDV3RcKm.

20. LECUN, Yann; CORTES, Corinna. MNIST handwritten digit database. 2010. Available also from: http://yann.lecun.com/exdb/mnist/.

21. KRIZHEVSKY, Alex; HINTON, Geoffrey, et al. Learning multiple layers of features from tiny images. *University of Toronto*. 2009. Available also from: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

22. CHOLLET, François et al. *Keras 2.2.4-tf* [comp. software]. 2015. Available also from: https://keras.io.

23. DAMELIN, S. B.; HOANG, N. S. On Surface Completion and Image Inpainting by Biharmonic Functions: Numerical Aspects. *International Journal of Mathematics and Mathematical Sciences* [online]. 2018, vol. 2018, pp. 1–8 [visited on 2020-04-27]. ISSN 1687-0425. Available from DOI: 10.1155/2018/3950312.

24. TELEA, Alexandru. An Image Inpainting Technique Based on the Fast Marching Method. *Journal of Graphics Tools*. 2004, vol. 9, no. 1, pp. 23–34. Available from DOI: 10.1080/10867651.2004.10487596.

25. LAHIRI, A.; JAIN, A. K.; NADENDLA, D.; BISWAS, P. K. Faster Unsupervised Semantic Inpainting: A GAN Based Approach. In: *2019 IEEE International Conference on Image Processing (ICIP)*. 2019, pp. 2706–2710. ISBN 978-1-5386-6249-6. ISSN 2381-8549. Available from DOI: 10.1109/ICIP.2019.8803356.

Appendix A

Acronyms

CE Context Encoder

CNN Convolutional Neural Network

GAIN Generative Adversarial Imputation Network

GAN Generative Adversarial Network

GPU Graphical Processing Unit

MSE Mean Squared Error

ReLU Rectified Linear Unit

RGB Red Green Blue (colour model)

SDA Stacked Denoising Autoencoder

SGD Stochastic Gradient Descent

SSDA Stacked Sparse Denoising Autoencoder

Appendix **B**

Contents of Enclosed SD card

```
readme.txt ........................ brief summary of the SD card's content
src ...................................... directory with implementation files
  training ............. IPython notebooks used in training the models
  evaluation ......... IPython notebooks used in evaluating the results
    mse ................................... mean-squared error evaluation
    classifier_accuracy ............ classifier training and evaluation
thesis_text ........................... directory with text of the thesis
  assignment.pdf ...................... the assignment in PDF format
  BP_Halama_Tomas_2020.pdf ................ this thesis in PDF format
```

Appendix C

Results Tables

Table C.1: MSE results for MNIST, centre square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.2076	0.5818	0.6939	0.7661
20%	0.3059	0.6570	0.7646	0.8822
30%	0.3597	0.6478	0.7870	0.8884
40%	0.4047	0.6366	0.7924	0.8704
50%	0.4543	0.5973	0.7091	0.8032

Table C.2: MSE results for MNIST, corner square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.0209	0.0431	0.0631	0.0921
20%	0.0870	0.1736	0.3139	0.2933
30%	0.1325	0.2397	0.4758	0.3680
40%	0.1820	0.2983	0.5915	0.4363
50%	0.2427	0.3488	0.6912	0.5030

Table C.3: MSE results for MNIST, random noise damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.0905	0.0871	0.0855	0.0750
20%	0.1812	0.1739	0.1701	0.1467
30%	0.2715	0.2597	0.2531	0.2154
40%	0.3618	0.3450	0.3349	0.2822
50%	0.4521	0.4293	0.4146	0.3468

Table C.4: MSE results for CIFAR-10, centre square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.1015	0.1919	0.0924	0.1088
20%	0.1172	0.1963	0.1188	0.1340
30%	0.1266	0.1986	0.1305	0.1448
40%	0.1375	0.1996	0.1496	0.1639
50%	0.1430	0.2007	0.1588	0.1725

Table C.5: MSE results for CIFAR-10, corner square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.1076	0.2187	0.1181	0.1047
20%	0.1319	0.2165	0.1755	0.1365
30%	0.1439	0.2159	0.2013	0.1480
40%	0.1634	0.2136	0.2407	0.1705
50%	0.1730	0.2129	0.2527	0.1833

Table C.6: MSE results for CIFAR-10, random noise damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.0213	0.0177	0.0244	0.0206
20%	0.0438	0.0345	0.0485	0.0400
30%	0.0665	0.0511	0.0724	0.0585
40%	0.0876	0.0676	0.0961	0.0762
50%	0.1054	0.0848	0.1195	0.0931

Table C.7: Classifier accuracy baseline values for undamaged datasets.

Dataset	Train	Validation	Test
MNIST	0.9970	0.9848	0.9866
CIFAR-10	0.6792	0.6912	0.6956

Table C.8: Classifier accuracy results for MNIST, centre square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.9675	0.8484	0.8377	0.8622
20%	0.9076	0.5617	0.6262	0.6207
30%	0.8316	0.4623	0.5062	0.4895
40%	0.7449	0.3572	0.3648	0.3611
50%	0.6166	0.2894	0.2508	0.3098

Table C.9: Classifier accuracy results for MNIST, corner square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.9866	0.9858	0.9856	0.9853
20%	0.9761	0.9596	0.9478	0.9370
30%	0.9631	0.9125	0.8845	0.8589
40%	0.9276	0.7696	0.7518	0.7371
50%	0.8520	0.5902	0.5584	0.5590

Table C.10: Classifier accuracy results for MNIST, random noise damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.9859	0.9860	0.9863	0.9855
20%	0.9858	0.9850	0.9860	0.9846
30%	0.9842	0.9839	0.9848	0.9813
40%	0.9833	0.9801	0.9841	0.9770
50%	0.9805	0.9739	0.9807	0.9688

Table C.11: Classifier accuracy results for CIFAR-10, centre square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.6392	0.5794	0.6519	0.6455
20%	0.5880	0.4915	0.5936	0.5809
30%	0.5467	0.4378	0.5467	0.5314
40%	0.4329	0.2739	0.3963	0.3690
50%	0.3660	0.2012	0.3207	0.2882

Table C.12: Classifier accuracy results for CIFAR-10, corner square damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.6783	0.6752	0.6832	0.6791
20%	0.6282	0.6024	0.6347	0.6320
30%	0.5912	0.5482	0.5942	0.5932
40%	0.4963	0.3888	0.5012	0.4887
50%	0.4277	0.2947	0.4257	0.4044

Table C.13: Classifier accuracy results for CIFAR-10, random noise damage.

Damage severity	Inpainting method			
	CE	GAIN	Biharmonic	Telea
10%	0.6684	0.6691	0.6936	0.6766
20%	0.6306	0.6294	0.6854	0.6463
30%	0.5943	0.5806	0.6739	0.5961
40%	0.5557	0.5176	0.6558	0.5398
50%	0.5140	0.4547	0.6267	0.4816

Appendix D

Visual Demonstration of Results

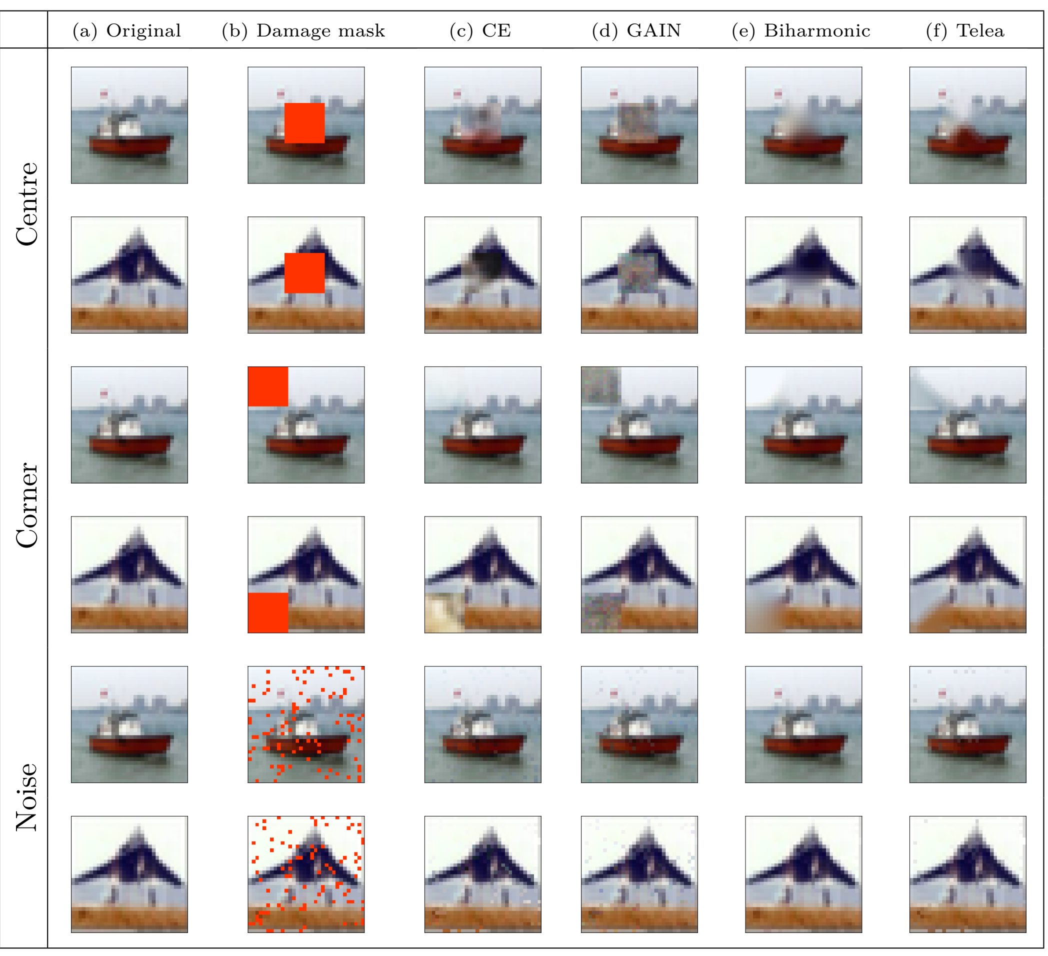


Table D.1: Demonstration of the inpainting results for all damage types and methods. Shown on 10% damaged CIFAR-10 dataset.

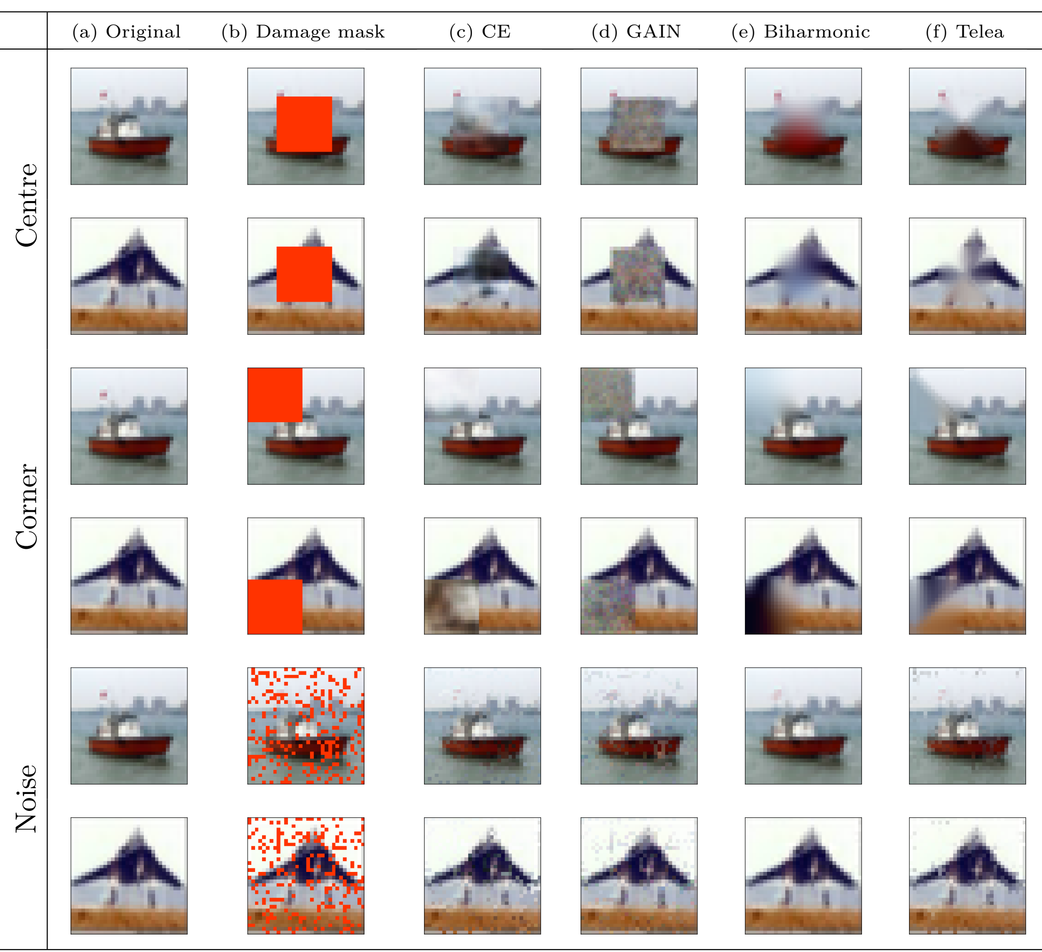


Table D.2: Demonstration of the inpainting results for all damage types and methods. Shown on 20% damaged CIFAR-10 dataset.

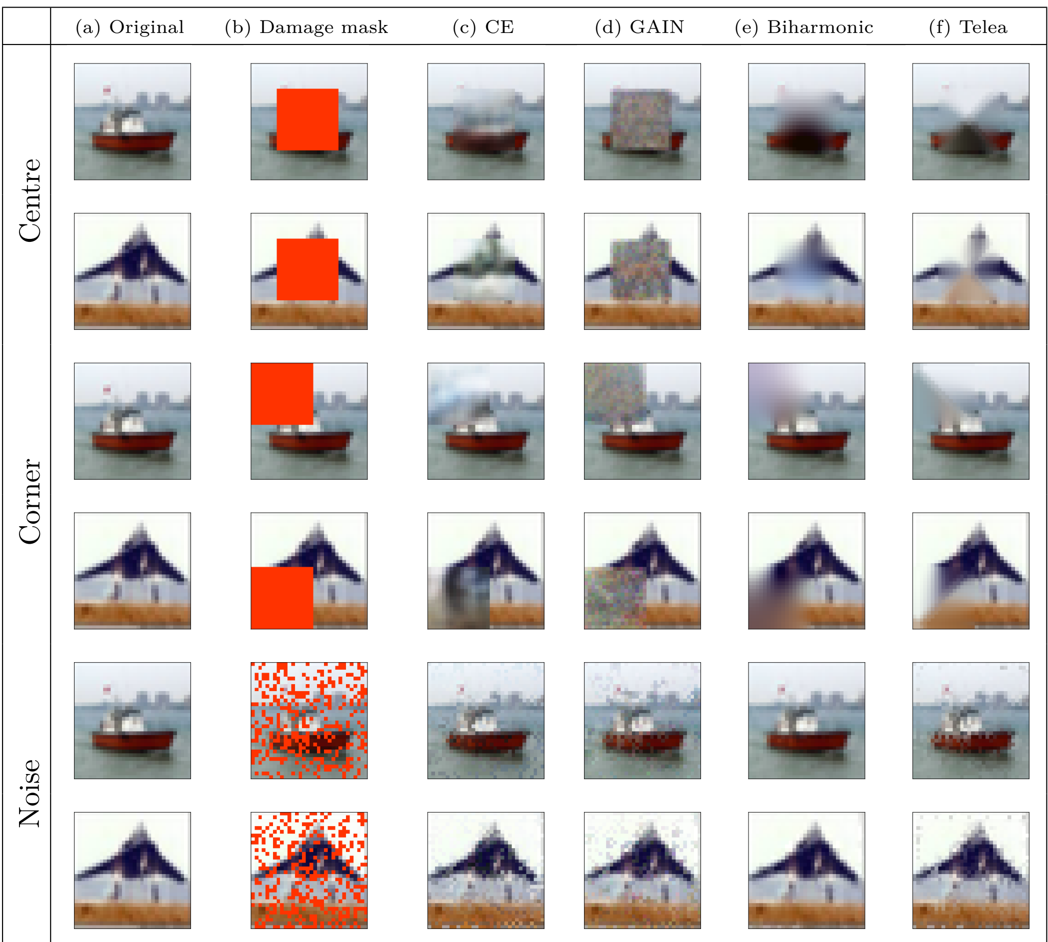


Table D.3: Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged CIFAR-10 dataset.

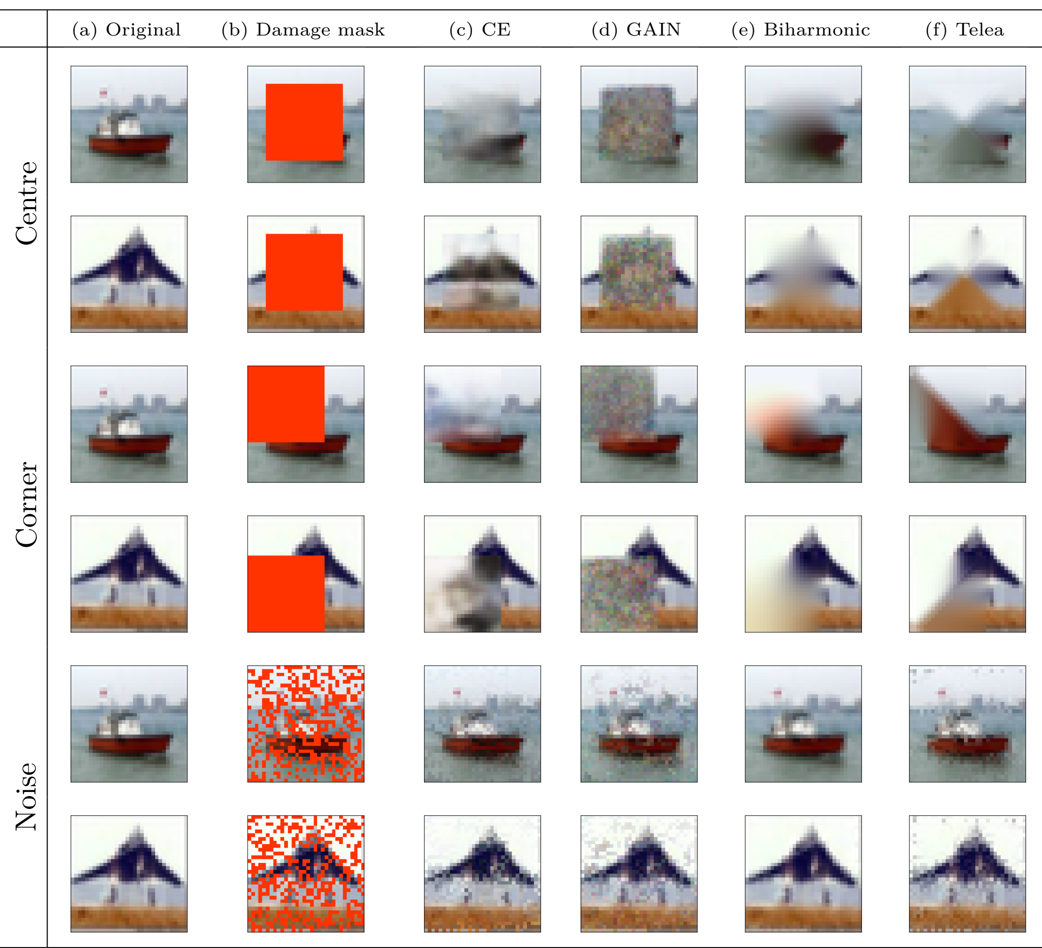


Table D.4: Demonstration of the inpainting results for all damage types and methods. Shown on 40% damaged CIFAR-10 dataset.

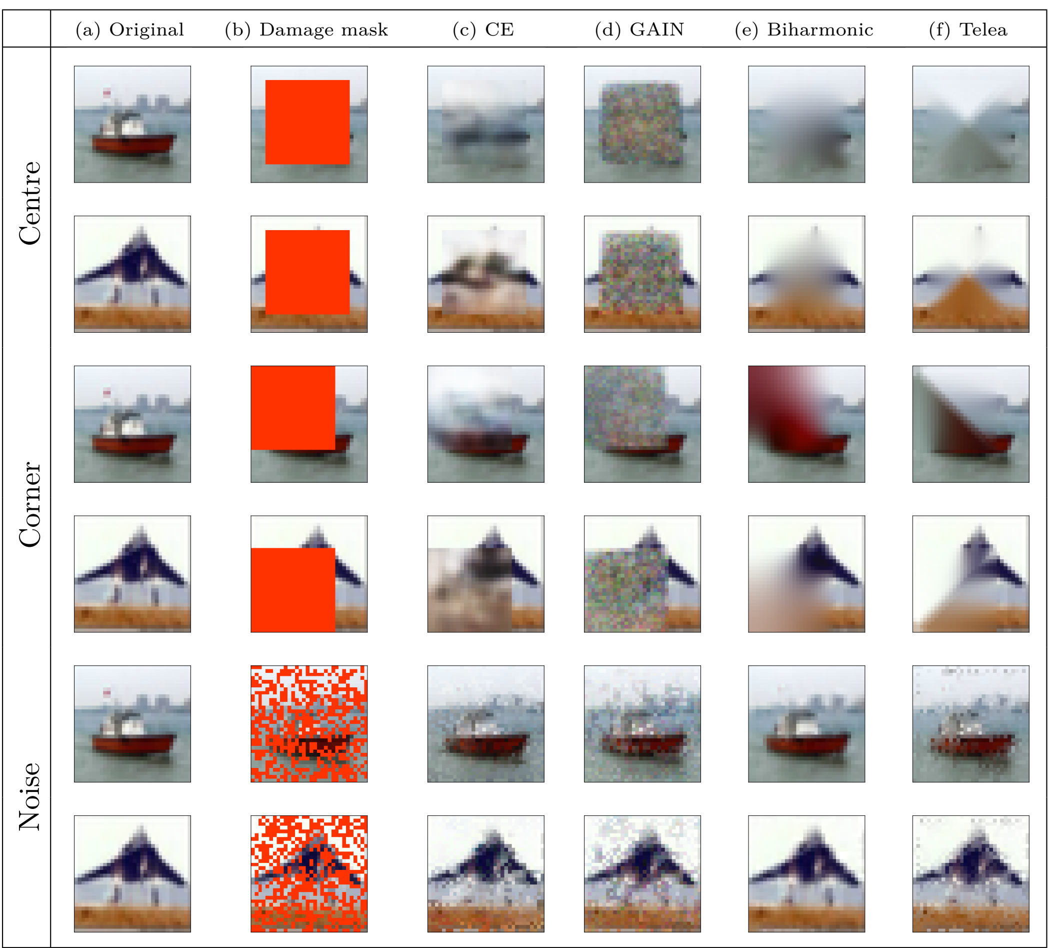


Table D.5: Demonstration of the inpainting results for all damage types and methods. Shown on 50% damaged CIFAR-10 dataset.

Table D.6: Demonstration of the inpainting results for all damage types and methods. Shown on 10% damaged MNIST dataset.

Table D.7: Demonstration of the inpainting results for all damage types and methods. Shown on 20% damaged MNIST dataset.

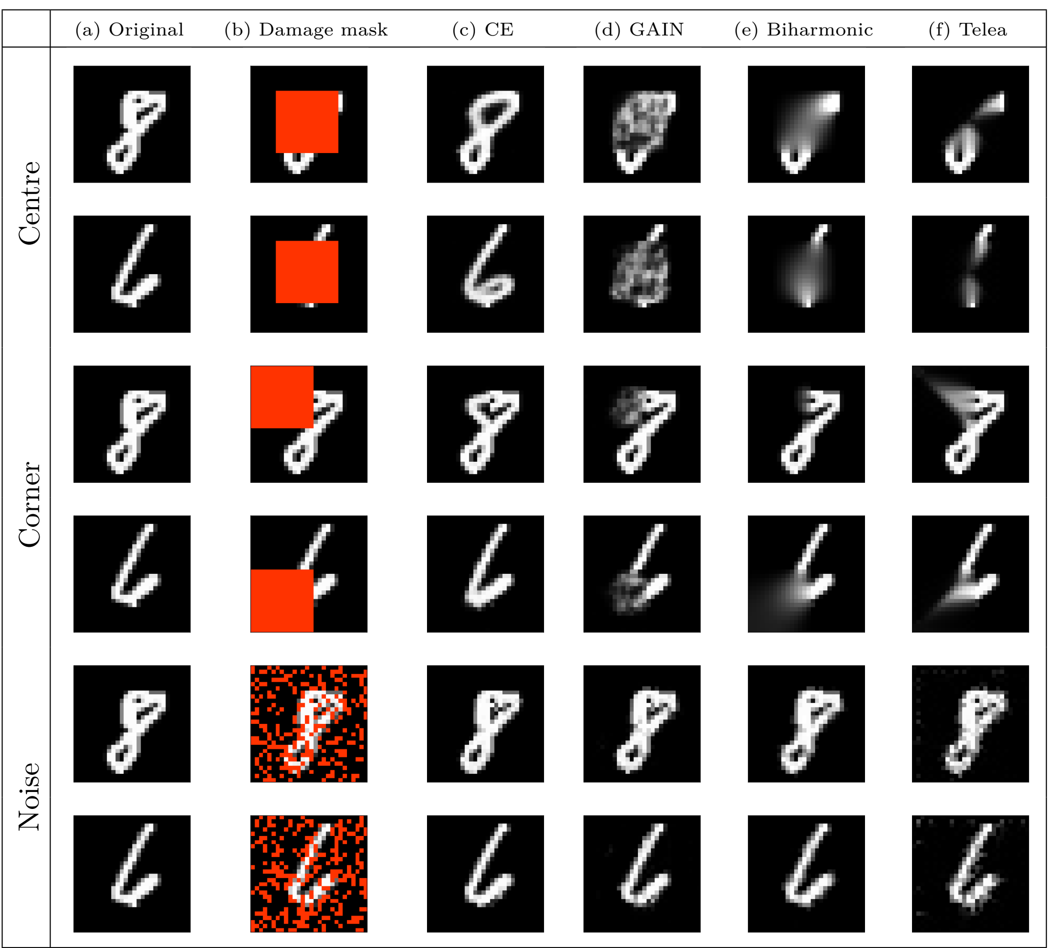


Table D.8: Demonstration of the inpainting results for all damage types and methods. Shown on 30% damaged MNIST dataset.

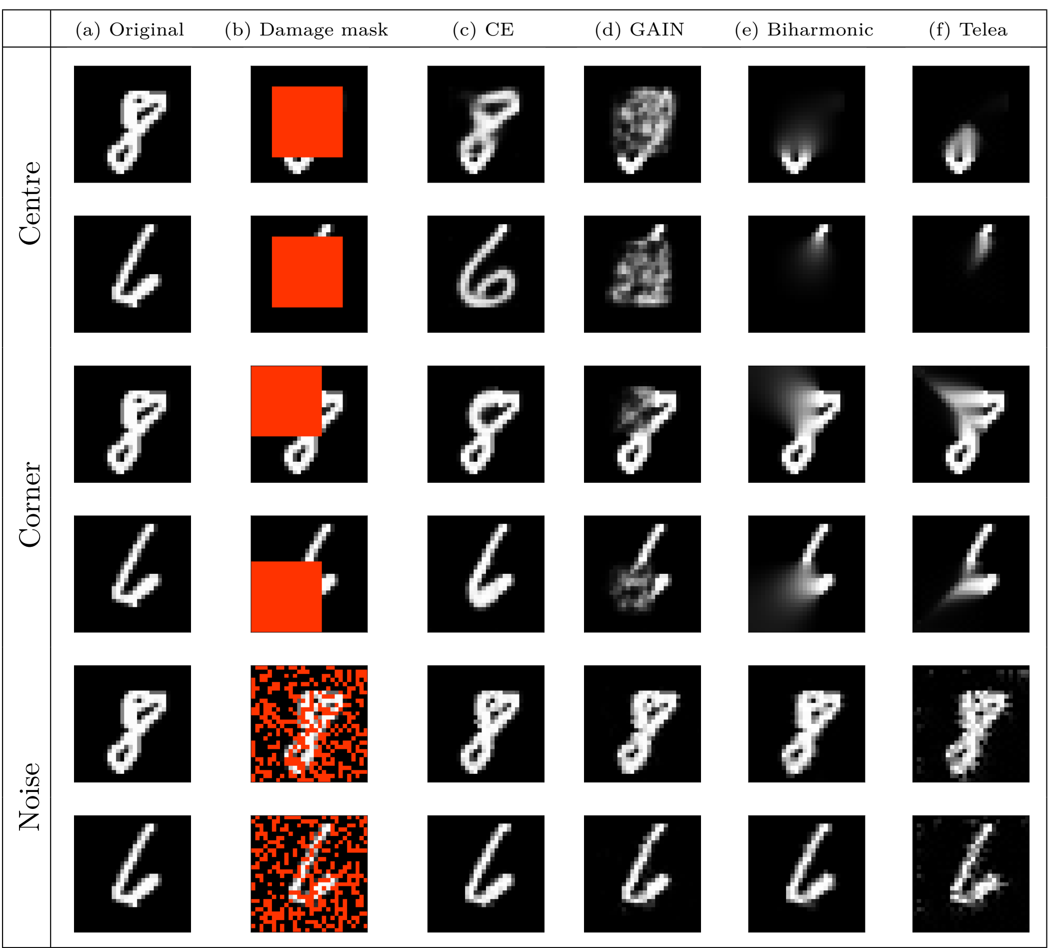


Table D.9: Demonstration of the inpainting results for all damage types and methods. Shown on 40% damaged MNIST dataset.

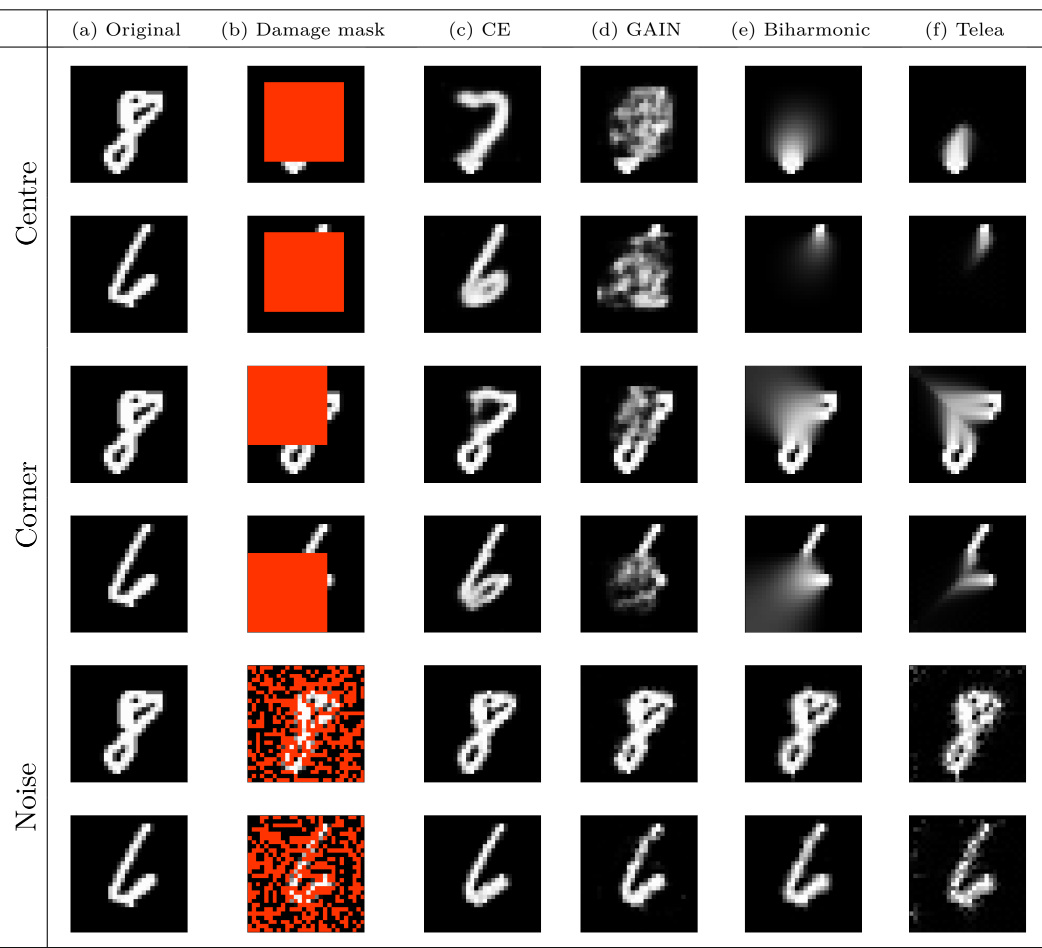


Table D.10: Demonstration of the inpainting results for all damage types and methods. Shown on 50% damaged MNIST dataset.

Appendix E

Network Architectures

The tables presented in this appendix chapter were generating using a part of the `keras-reports`[12] framework.

[12] https://github.com/fablukm/keras-reports

Table E.1: Context Encoder discriminator, 50% centre damaged CIFAR-10.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	forged_real_input (InputLayer)	(23, 23, 3)		0	
1	conv2d_24 (Conv2D)	(12, 12, 32)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	896	forged_real_input
2	batch_normalization_v1_40 (BatchNormalization)	(12, 12, 32)		128	conv2d_24
3	leaky_re_lu_24 (LeakyReLU)	(12, 12, 32)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_40
4	dropout_8 (Dropout)	(12, 12, 32)	**Dropout Rate:** 0.3	0	leaky_re_lu_24
5	conv2d_25 (Conv2D)	(6, 6, 64)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	18 496	dropout_8
6	batch_normalization_v1_41 (BatchNormalization)	(6, 6, 64)		256	conv2d_25
7	leaky_re_lu_25 (LeakyReLU)	(6, 6, 64)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_41
8	dropout_9 (Dropout)	(6, 6, 64)	**Dropout Rate:** 0.3	0	leaky_re_lu_25
9	conv2d_26 (Conv2D)	(3, 3, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	73 856	dropout_9
10	batch_normalization_v1_42 (BatchNormalization)	(3, 3, 128)		512	conv2d_26
11	leaky_re_lu_26 (LeakyReLU)	(3, 3, 128)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_42
12	dropout_10 (Dropout)	(3, 3, 128)	**Dropout Rate:** 0.3	0	leaky_re_lu_26
13	conv2d_27 (Conv2D)	(2, 2, 256)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	295 168	dropout_10
14	batch_normalization_v1_43 (BatchNormalization)	(2, 2, 256)		1024	conv2d_27
15	leaky_re_lu_27 (LeakyReLU)	(2, 2, 256)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_43
16	dropout_11 (Dropout)	(2, 2, 256)	**Dropout Rate:** 0.3	0	leaky_re_lu_27
17	flatten_1 (Flatten)	(1024,)		0	dropout_11
18	dense_5 (Dense)	(1,)	**#Neurons:** 1 **Activation:** sigmoid	1025	flatten_1

Table E.2: Context Encoder autoencoder, 50% centre damaged CIFAR-10, continued in Table E.3.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	damaged_input (InputLayer)	(32, 32, 3)		0	
1	conv2d (Conv2D)	(29, 29, 32)	**Activation:** linear **Kernel Size:** [4, 4] **Stride:** [1, 1]	1568	damaged_input
2	batch_normalization_v1 (BatchNormalization)	(29, 29, 32)		128	conv2d
3	leaky_re_lu (LeakyReLU)	(29, 29, 32)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1
4	max_pooling2d (MaxPooling2D)	(9, 9, 32)	**Pool size:** [3, 3] **Strides:** [3, 3]	0	leaky_re_lu
5	conv2d_1 (Conv2D)	(9, 9, 64)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	18 496	max_pooling2d
6	batch_normalization_v1_1 (BatchNormalization)	(9, 9, 64)		256	conv2d_1
7	leaky_re_lu_1 (LeakyReLU)	(9, 9, 64)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_1
8	conv2d_2 (Conv2D)	(7, 7, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	73 856	leaky_re_lu_1
9	batch_normalization_v1_2 (BatchNormalization)	(7, 7, 128)		512	conv2d_2
10	leaky_re_lu_2 (LeakyReLU)	(7, 7, 128)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_2
11	max_pooling2d_1 (MaxPooling2D)	(3, 3, 128)	**Pool size:** [2, 2] **Strides:** [2, 2]	0	leaky_re_lu_2
12	conv2d_3 (Conv2D)	(3, 3, 256)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	295 168	max_pooling2d_1
13	batch_normalization_v1_3 (BatchNormalization)	(3, 3, 256)		1024	conv2d_3
14	leaky_re_lu_3 (LeakyReLU)	(3, 3, 256)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_3
15	conv2d_4 (Conv2D)	(1, 1, 512)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	1 180 160	leaky_re_lu_3

Table E.3: Context Encoder autoencoder, 50% centre damaged CIFAR-10, continuation of Table E.2.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
16	batch_normalization_v1_4 (BatchNormalization)	(1, 1, 512)		2048	conv2d_4
17	leaky_re_lu_4 (LeakyReLU)	(1, 1, 512)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_4
18	dense (Dense)	(1, 1, 512)	**#Neurons:** 512 **Activation:** linear	262 656	leaky_re_lu_4
19	dropout (Dropout)	(1, 1, 512)	**Dropout Rate:** 0.3	0	dense
20	conv2d_transpose (Conv2DTranspose)	(3, 3, 256)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	1 179 904	dropout
21	batch_normalization_v1_5 (BatchNormalization)	(3, 3, 256)		1024	conv2d_transpose
22	re_lu (ReLU)	(3, 3, 256)	**Activation:** relu	0	batch_normalization_v1_5
23	conv2d_transpose_1 (Conv2DTranspose)	(7, 7, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	295 040	re_lu
24	batch_normalization_v1_6 (BatchNormalization)	(7, 7, 128)		512	conv2d_transpose_1
25	re_lu_1 (ReLU)	(7, 7, 128)	**Activation:** relu	0	batch_normalization_v1_6
26	conv2d_transpose_2 (Conv2DTranspose)	(16, 16, 64)	**Activation:** linear **Kernel Size:** [4, 4] **Stride:** [2, 2]	131 136	re_lu_1
27	batch_normalization_v1_7 (BatchNormalization)	(16, 16, 64)		256	conv2d_transpose_2
28	re_lu_2 (ReLU)	(16, 16, 64)	**Activation:** relu	0	batch_normalization_v1_7
29	conv2d_transpose_3 (Conv2DTranspose)	(19, 19, 32)	**Activation:** linear **Kernel Size:** [4, 4] **Stride:** [1, 1]	32 800	re_lu_2
30	batch_normalization_v1_8 (BatchNormalization)	(19, 19, 32)		128	conv2d_transpose_3
31	re_lu_3 (ReLU)	(19, 19, 32)	**Activation:** relu	0	batch_normalization_v1_8
32	conv2d_transpose_4 (Conv2DTranspose)	(23, 23, 3)	**Activation:** tanh **Kernel Size:** [5, 5] **Stride:** [1, 1]	2403	re_lu_3

Table E.4: Context Encoder discriminator, noise damaged CIFAR-10.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	forged_real_input (InputLayer)	(32, 32, 3)		0	
1	conv2d_5 (Conv2D)	(16, 16, 32)	**Activation:** linear **Kernel Size:** [5, 5] **Stride:** [2, 2]	2432	forged_real_input
2	batch_normalization_v1_10 (BatchNormalization)	(16, 16, 32)		128	conv2d_5
3	leaky_re_lu_5 (LeakyReLU)	(16, 16, 32)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_10
4	dropout_1 (Dropout)	(16, 16, 32)	**Dropout Rate:** 0.3	0	leaky_re_lu_5
5	conv2d_6 (Conv2D)	(8, 8, 64)	**Activation:** linear **Kernel Size:** [5, 5] **Stride:** [2, 2]	51 264	dropout_1
6	batch_normalization_v1_11 (BatchNormalization)	(8, 8, 64)		256	conv2d_6
7	leaky_re_lu_6 (LeakyReLU)	(8, 8, 64)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_11
8	dropout_2 (Dropout)	(8, 8, 64)	**Dropout Rate:** 0.3	0	leaky_re_lu_6
9	conv2d_7 (Conv2D)	(4, 4, 64)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	36 928	dropout_2
10	batch_normalization_v1_12 (BatchNormalization)	(4, 4, 64)		256	conv2d_7
11	leaky_re_lu_7 (LeakyReLU)	(4, 4, 64)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_12
12	dropout_3 (Dropout)	(4, 4, 64)	**Dropout Rate:** 0.3	0	leaky_re_lu_7
13	conv2d_8 (Conv2D)	(2, 2, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	73 856	dropout_3
14	batch_normalization_v1_13 (BatchNormalization)	(2, 2, 128)		512	conv2d_8
15	leaky_re_lu_8 (LeakyReLU)	(2, 2, 128)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_13
16	dropout_4 (Dropout)	(2, 2, 128)	**Dropout Rate:** 0.3	0	leaky_re_lu_8
17	flatten (Flatten)	(512,)		0	dropout_4
18	dense_1 (Dense)	(1,)	**#Neurons:** 1 **Activation:** sigmoid	513	flatten

Table E.5: Context Encoder autoencoder, noise damaged CIFAR-10, continued in Table E.6.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	damaged_input (InputLayer)	(32, 32, 3)		0	
1	conv2d (Conv2D)	(29, 29, 32)	**Activation:** linear **Kernel Size:** [4, 4] **Stride:** [1, 1]	1568	damaged_input
2	batch_normalization_v1 (BatchNormalization)	(29, 29, 32)		128	conv2d
3	leaky_re_lu (LeakyReLU)	(29, 29, 32)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1
4	max_pooling2d (MaxPooling2D)	(9, 9, 32)	**Pool size:** [3, 3] **Strides:** [3, 3]	0	leaky_re_lu
5	conv2d_1 (Conv2D)	(9, 9, 64)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	18 496	max_pooling2d
6	batch_normalization_v1_1 (BatchNormalization)	(9, 9, 64)		256	conv2d_1
7	leaky_re_lu_1 (LeakyReLU)	(9, 9, 64)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_1
8	conv2d_2 (Conv2D)	(7, 7, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	73 856	leaky_re_lu_1
9	batch_normalization_v1_2 (BatchNormalization)	(7, 7, 128)		512	conv2d_2
10	leaky_re_lu_2 (LeakyReLU)	(7, 7, 128)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_2
11	max_pooling2d_1 (MaxPooling2D)	(3, 3, 128)	**Pool size:** [2, 2] **Strides:** [2, 2]	0	leaky_re_lu_2
12	conv2d_3 (Conv2D)	(3, 3, 256)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	295 168	max_pooling2d_1
13	batch_normalization_v1_3 (BatchNormalization)	(3, 3, 256)		1024	conv2d_3
14	leaky_re_lu_3 (LeakyReLU)	(3, 3, 256)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_3
15	conv2d_4 (Conv2D)	(1, 1, 512)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	1 180 160	leaky_re_lu_3

Table E.6: Context Encoder autoencoder, noise damaged CIFAR-10, continuation of Table E.5.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
16	batch_normalization_v1_4 (BatchNormalization)	(1, 1, 512)		2048	conv2d_4
17	leaky_re_lu_4 (LeakyReLU)	(1, 1, 512)	**Activation:** leakyrelu **Alpha:** 0.3	0	batch_normalization_v1_4
18	dense (Dense)	(1, 1, 512)	**#Neurons:** 512 **Activation:** linear	262 656	leaky_re_lu_4
19	dropout (Dropout)	(1, 1, 512)	**Dropout Rate:** 0.3	0	dense
20	conv2d_transpose (Conv2DTranspose)	(3, 3, 512)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	2 359 808	dropout
21	batch_normalization_v1_5 (BatchNormalization)	(3, 3, 512)		2048	conv2d_transpose
22	re_lu (ReLU)	(3, 3, 512)	**Activation:** relu	0	batch_normalization_v1_5
23	conv2d_transpose_1 (Conv2DTranspose)	(7, 7, 256)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	1 179 904	re_lu
24	batch_normalization_v1_6 (BatchNormalization)	(7, 7, 256)		1024	conv2d_transpose_1
25	re_lu_1 (ReLU)	(7, 7, 256)	**Activation:** relu	0	batch_normalization_v1_6
26	conv2d_transpose_2 (Conv2DTranspose)	(15, 15, 128)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	295 040	re_lu_1
27	batch_normalization_v1_7 (BatchNormalization)	(15, 15, 128)		512	conv2d_transpose_2
28	re_lu_2 (ReLU)	(15, 15, 128)	**Activation:** relu	0	batch_normalization_v1_7
29	conv2d_transpose_3 (Conv2DTranspose)	(30, 30, 64)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [2, 2]	73 792	re_lu_2
30	batch_normalization_v1_8 (BatchNormalization)	(30, 30, 64)		256	conv2d_transpose_3
31	re_lu_3 (ReLU)	(30, 30, 64)	**Activation:** relu	0	batch_normalization_v1_8
32	conv2d_transpose_4 (Conv2DTranspose)	(30, 30, 32)	**Activation:** linear **Kernel Size:** [3, 3] **Stride:** [1, 1]	18 464	re_lu_3
33	batch_normalization_v1_9 (BatchNormalization)	(30, 30, 32)		128	conv2d_transpose_4
34	re_lu_4 (ReLU)	(30, 30, 32)	**Activation:** relu	0	batch_normalization_v1_9
35	conv2d_transpose_5 (Conv2DTranspose)	(32, 32, 3)	**Activation:** tanh **Kernel Size:** [3, 3] **Stride:** [1, 1]	867	re_lu_4

Table E.7: GAIN generator, CIFAR-10.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	data (InputLayer)	(32, 32, 3)		0	
1	mask (InputLayer)	(32, 32, 1)		0	
2	random (InputLayer)	(32, 32, 1)		0	
3	concatenate (Concatenate)	(32, 32, 5)	**Axis:** -1	0	data, mask, random
4	flatten (Flatten)	(5120,)		0	concatenate
5	dense (Dense)	(3072,)	**#Neurons:** 3072 **Activation:** linear	15 731 712	flatten
	activation (Activation)	(3072,)	**Activation:** tanh	0	dense
6	dense_1 (Dense)	(1536,)	**#Neurons:** 1536 **Activation:** linear	4 720 128	activation
	activation_1 (Activation)	(1536,)	**Activation:** tanh	0	dense_1
7	dense_2 (Dense)	(3072,)	**#Neurons:** 3072 **Activation:** tanh	4 721 664	activation_1
8	reshape (Reshape)	(32, 32, 3)		0	dense_2

Table E.8: GAIN discriminator, CIFAR-10.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	imputed (InputLayer)	(32, 32, 3)		0	
1	hint (InputLayer)	(32, 32, 1)		0	
2	concatenate_1 (Concatenate)	(32, 32, 4)	**Axis:** -1	0	imputed, hint
3	flatten_1 (Flatten)	(4096,)		0	concatenate_1
4	dense_3 (Dense)	(3072,)	**#Neurons:** 3072 **Activation:** linear	12 585 984	flatten_1
	activation_2 (Activation)	(3072,)	**Activation:** tanh	0	dense_3
5	dense_4 (Dense)	(512,)	**#Neurons:** 512 **Activation:** linear	1 573 376	activation_2
	activation_3 (Activation)	(512,)	**Activation:** tanh	0	dense_4
6	dense_5 (Dense)	(1024,)	**#Neurons:** 1024 **Activation:** sigmoid	525 312	activation_3
7	reshape_1 (Reshape)	(32, 32, 1)		0	dense_5

Table E.9: GAIN generator, MNIST.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	data (InputLayer)	(28, 28, 1)		0	
1	mask (InputLayer)	(28, 28, 1)		0	
2	random (InputLayer)	(28, 28, 1)		0	
3	concatenate (Concatenate)	(28, 28, 3)	**Axis:** -1	0	data, mask, random
4	flatten (Flatten)	(2352,)		0	concatenate
5	dense (Dense)	(784,)	**#Neurons:** 784 **Activation:** linear	1 844 752	flatten
	activation (Activation)	(784,)	**Activation:** tanh	0	dense
6	dense_1 (Dense)	(392,)	**#Neurons:** 392 **Activation:** linear	307 720	activation
	activation_1 (Activation)	(392,)	**Activation:** tanh	0	dense_1
7	dense_2 (Dense)	(784,)	**#Neurons:** 784 **Activation:** tanh	308 112	activation_1
8	reshape (Reshape)	(28, 28, 1)		0	dense_2

Table E.10: GAIN discriminator, MNIST.

№	Layer (Type)	Output shape	Config	#Parameters	Inbound layers
0	imputed (InputLayer)	(28, 28, 1)		0	
1	hint (InputLayer)	(28, 28, 1)		0	
2	concatenate_1 (Concatenate)	(28, 28, 2)	**Axis:** -1	0	imputed, hint
3	flatten_1 (Flatten)	(1568,)		0	concatenate_1
4	dense_3 (Dense)	(784,)	**#Neurons:** 784 **Activation:** linear	1 230 096	flatten_1
	activation_2 (Activation)	(784,)	**Activation:** tanh	0	dense_3
5	dense_4 (Dense)	(392,)	**#Neurons:** 392 **Activation:** linear	307 720	activation_2
	activation_3 (Activation)	(392,)	**Activation:** tanh	0	dense_4
6	dense_5 (Dense)	(784,)	**#Neurons:** 784 **Activation:** sigmoid	308 112	activation_3
7	reshape_1 (Reshape)	(28, 28, 1)		0	dense_5