BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

RECONSTRUCTION OF MISSING PARTS OF THE FACE USING NEURAL NETWORK

REKONSTRUKCE CHYBĚJÍCÍCH ČÁSTI OBLIČEJE POMOCÍ NEURONOVÉ SÍTĚ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN MAREK**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ GOLDMANN**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

Student: **Marek Jan, Bc.**
Programme: Information Technology Field of study: Intelligent Systems
Title: **Reconstruction of Missing Parts of the Face Using Neural Network**
Category: Artificial Intelligence

Assignment:

1. Familiarize yourself with the techniques used for face recognition and find out which aspects influence the success rate of these methods the most.
2. Familiarize yourself with the theory behind neural networks, especially convolutional neural networks (CNN) and generative adversarial networks (GAN).
3. Based on existing solutions, design a model of a neural network for the reconstruction of missing or damaged areas of a photo of a face.
4. Implement and train the proposed neural network models in Python programming language.
5. Perform experiments and evaluate the performance of the model based on ground truth images. Prepare a dataset containing at least 100 images of people of different races and genders. Adjust the model based on obtained results to achieve the best possible performance.

Recommended literature:

- YU, Jiahui, et al. Free-form image inpainting with gated convolution. *arXiv preprint arXiv:1806.03589*, 2018.
- RADFORD, Alec; METZ, Luke; CHINTALA, Soumith. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.
- LI, Xiaoming, et al. Learning Symmetry Consistent Deep CNNs for Face Completion. *arXiv preprint arXiv:1812.07741*, 2018.

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Goldmann Tomáš, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: April 15, 2020

Abstract

The goal of this thesis is to design a neural network for reconstruction of face images in which a part of the face is obscured by a mask. Concepts used in the development of convolutional neural networks and generative adversarial networks are presented. Specific concepts used in neural networks used for face reconstruction are described. The generative adversarial network presented in this thesis combines the use of gated convolutional layers and dense multiscale fusion blocks to produce realistic reconstructions of masked face images.

Abstrakt

Cílem této práce je vytvořit neuronovou síť která bude schopna rekonstruovat obličeje z fotografií na kterých je část obličeje překrytá maskou. Jsou prezentovány koncepty využívané při vývoji konvolučních neuronových sítí a generativních kompetitivních sítí. Dále jsou popsány koncepty používané v neuronových sítích specificky pro rekonstrukci fotografií obličejů. Je představen model generativní kompetitivní sítě využívající kombinaci hrazených konvolučních vrstev a víceškálových bloků schopný realisticky doplnit oblasti obličeje zakryté maskou.

Keywords

face image reconstruction, image reconstruction, face image inpainting, image inpainting, neural network, convolutional neural network, CNN, generative adversarial network, GAN, machine learning, gated convolution, dense multiscale fusion block, DMFB, spectral normalization

Klíčová slova

rekonstrukce snímku obličeje, rekonstrukce snímku, neuronová síť, konvoluční neuronová síť, CNN, generativní kompetitivní síť, GAN, strojové učení, hrazené konvoluce, spektrální normalizace

Reference

MAREK, Jan. *Reconstruction of Missing Parts of the Face Using Neural Network.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Goldmann

Rozšířený abstrakt

Cílem této diplomové práce je vytvoření neuronové sítě pro rekonstrukci chybějících částí obličeje. Rekonstrukce obličeje je úkol řešený v rámci strojového učení kde cílem je co nejpřesněji opravit snímek obličeje ve kterém je část obličeje smazána či překryta maskou. Lidský obličej obsahuje mnoho rysů, na které jsme velmi citliví a na nichž z pohledu lidského vnímání závisí rozpoznání dané osoby. Tyto rysy tvoří jak malé detaily jako barva očí, ale i velké součásti jako například tvar nosu nebo celkový tvar obličeje. Z tohoto důvodu je rekonstrukce obličeje poměrně těžký úkol, co se přesnosti výsledků týče, protože i když je v opraveném obličeji pouze malá změna, může daný člověk vypadat zásadně jinak. Neuronové sítě využívané pro rekonstrukci obličeje extrahují informace z části obličeje, která není překryta maskou, tyto informace zpracují a vychází z nich tak aby dokreslily celý obličej co nejpřesněji. Nemělo by se tedy například stát, že by člověku s černými vlasy bylo dokresleno blond obočí. Dále by měly být dokreslené oblasti obličeje co nejpřesnější co se týče etnicity, věku a pohlaví dané osoby. Tyto neuronové sítě tedy na základě naučených informací o tom, jak lidské obličeje vypadají zkouší dokreslit nejpravděpodobnější obličej na základě dostupných informací. Dokreslený obličej ale ze samotné podstaty problému nemůže být perfektní, neboť každý člověk je jiný a síť pouze usuzuje na základě naučených obecných vlastností obličejů. Cílem tedy není stoprocentní přesnost doplněného obličeje, ale co nejvyšší realističnost dokresleného obličeje.

Úkol rekonstrukce obličeje se řadí do třídy úkolů zabývající se zpracováním obrazu. Tyto úkoly jsou v současnosti nejčastěji řešeny za pomoci konvolučních neuronových sítí. Tento typ sítí umožňuje efektivní zpracování obrazových vstupů za pomoci konvolučních vrstev. Tyto vrstvy transformují vstup pomocí konvolučních filtrů a jejich výstupem je tzv. příznaková mapa. Konvoluční filtr je dvourozměrné pole vah, které je postupně posouváno po vstupním obraze. V každé pozici je spočítán vážený součet vstupních pixelů, vstupní pixely jsou vynásobeny vahami a následně sečteny. Tento součet je pak hodnotou výstupního pixelu pro danou pozici filtru. Pohybováním filtru po celé ploše vstupního obrazu je pak vytvořena kompletní příznaková mapa. Váhy konvolučních filtrů jsou s pomocí trénovacího algoritmu měněny tak, aby byl výstup neuronové sítě co nejblíže k cílovému výstupu. S pomocí neuronových sítí je tedy možné vytvořit modely sítí pro extrakci informací z obrazových vstupů, či pro transformaci vstupního obrazu na jiný výstupní. Problém nastává při využití klasických konvolučních sítí pro rekonstrukci obrazu, klasická konvoluční síť totiž počítá s tím, že všechny oblasti ve vstupním obraze budou mít nějaký význam. To neplatí pro vstupní data sítí pro rekonstrukci obrazu, v takovém vstupu je určitá část překryta maskou, která se většinou reprezentuje čistě bílou barvou. Oblasti překryté maskou tedy neobsahují žádné důležité informace pro rekonstrukci, ale i přesto jsou zpracovávány stejným konvolučním filtrem jako oblasti které důležité informace skutečně obsahují. Tento problém byl vyřešen v roce 2018 za pomoci hrazených konvolučních vrstev (gated convolutions), tyto konvoluční vrstvy obsahují mechanismus, za jehož pomoci je extrahována ze vstupu maska reprezentující důležitost daných oblastí ve vstupním obrázku. Celá příznaková mapa je pak vynásobena touto maskou, je tím tedy vyřešen problém oblastí s bezvýznamnými daty.

Dalším významným konceptem v oblasti zpracování obrazu jsou generativní kompetitivní sítě. Tyto sítě jsou určeny pro generování nových dat na základě sady trénovacích dat. Kompetitivní síť sestává ze dvou součástí, generátoru a diskriminátoru. Cílem generátoru je vytvořit výstup, který je podobný datům z trénovací sady ale přímo do nich nepatří. Cílem diskriminátoru je rozlišit, jestli je jeho obrázek na jeho vstupu vytvořen generátorem, nebo jestli patří do trénovací sady dat. Tyto dvě sítě mají prakticky dva rozlišné úkoly, úkolem generátoru je ošálit diskriminátor a úkolem diskriminátoru je co nejpřesněji rozlišit

zdroj jeho vstupu. Výsledkem této soutěže je to že se generátor naučí vytvářet věrohodně vypadající data. V případě této práce je úkolem generátoru vytvořit dokreslený obličej na základě poškozeného vstupu.

V roce 2020 byl speciálně pro rekonstrukci obrazu navržen koncept víceškálového bloku (dense multiscale fusion block). Cílem tohoto bloku je efektivní zpracování dat z co nejširšího kontextu pro daný pixel, což je velmi důležité v kontextu rekonstrukce obličejů. Pro věrohodné dokreslení dat do maskované oblasti obličeje je důležité zpracování dat z co nejširšího okolí zamaskovaného pixelu. Kombinace multiškálových bloků a hrazených konvolucí nebyla nikdy vyzkoušena i přestože byl koncept víceškálových bloků publikován po představení hrazených konvolucí.

V rámci této práce byla úspěšně vytvořena kompetitivní generativní síť využívající hrazené konvoluční vrstvy v kombinaci s víceškálovými bloky pro rekonstrukci chybějících částí obličejů. Architektura víceškálových bloků byla upravena tak aby lépe využívala vlastností hrazených konvolučních vrstev. Neuronová síť, ve které byla využita upravená varianta víceškálových bloků dosahuje vyšší přesnosti a kvality výstupních rekonstrukcí než varianta s neupravenými bloky. Síť využívající při rekonstrukci bloky s původní architekturou dosahuje průměrné absolutní chyby 0,07 a PSNR 17,7. Rekonstruované fotky poškozených obličejů vytvořené sítí využívající upravené víceškálové bloky dosahují průměrné absolutní chyby **0,056** a PSNR **20,35** za stejných trénovací podmínek. Výsledná síť je schopna věrně dokreslit poškozené oblasti snímku obličeje.

Reconstruction of Missing Parts of the Face Using Neural Network

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Tomáš Goldmann. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jan Marek
July 30, 2020

Acknowledgements

I would like to thank my supervisor Mr. Tomáš Goldmann for his guidance and support in creating this thesis. I would also like to thank my family for their support. Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Contents

Chapter 1

Introduction

Neural networks are currently one of the most rapidly evolving areas of computer science. Neural networks have in the recent past become capable of generating high quality images including face images. This was made possible with the introduction of generative adversarial networks. Apart from just generating new faces, these networks have also been used for face reconstruction. Face reconstruction networks attempt to repair images in which some parts of the face are obscured. These networks extract information from images and try to use this information to fill in the missing parts. Face reconstruction is particularly difficult when compared to other applications of image reconstruction because the human face contains a high amount of information. Even a small change in the face image could change the appearance of the person in question enough to make them look like someone else. Networks used for face reconstruction are basically making an educated guess as to what the person in question looks like so the reconstructed face can look slightly different. However the goal is for the reconstruction to be as close as possible to the original person. A network performing face reconstruction can for example be used to find out what a person could look like from a photo in which their face is partially covered by a mask or sunglasses. Face reconstruction can also be used in forensic circumstances. A well-working face reconstruction network could for example be used to aid in identifying a suspect of a crime who attempted to conceal their identity by wearing a mask.

The next chapter will give the necessary theoretical basis needed for creating a neural network able to successfully reconstruct images. Firstly, methods for face recognition and face detection are presented. Neural networks are presented along with all commonly components and algorithms used for training. Finally, types of neural networks commonly used for face reconstruction are discussed along with concepts developed specifically for face reconstruction.

Chapter 3 contains information about the proposed model and implementation details. Available training datasets are presented and compared. Next an overview of the tools which were used for the implementation and training of the proposed model. Finally, an in-depth description of the architecture of the proposed model is given.

Chapter 4 provides a description of three models whose performance is then evaluated. Evaluation metrics which were used are presented. The models are then compared in terms of performance based on metrics and a visual assessment of their performance is also given.

Chapter 2

Theory

This chapter will briefly introduce the tasks of face recognition and face reconstruction as well as the techniques used to achieve both. Later on neural network architectures used specifically for face reconstruction will be introduced including all relevant information related to them.

2.1 Face recognition

This section is based on [39]. Face recognition is a task which falls into the category of computer vision. The purpose of facial recognition is to extract meaningful data from an image or a video and use that data to identify the person in the source material. This is not an easy task by any means, since even humans generally struggle differentiating between unknown faces. Face recognition algorithms have however had considerable success in this area.

This task can be separated into several steps, for a face to be recognized by the system and assigned to a specific person, the face needs to be identified in the source material itself.

2.1.1 Face detection

Face detection algorithms are a family of algorithms using various approaches to localize a face in an image or video source. These algorithms belong to a broader family of object detection algorithms, which search for a so-called region of interest in the source material. The output of these algorithms is generally speaking an area within the source material containing the desired object, which is in the case of face detection algorithms the face.

Before applying these algorithms, the data is typically preprocessed in order to increase the effectiveness of these algorithms. These preprocessing techniques typically include histogram equalization, colour transformations and filtering or downsampling of the source material.

Viola–Jones The Viola–Jones algorithm is a machine learning approach to object detection. This algorithm was published in 2001 by Paul Viola and Michael Jones [43]. The Viola–Jones algorithm made use of Haar features and introduced the Integral Image representation which made the computation significantly faster and more accurate than previously known algorithms at the time of publication. This algorithm has a very high detection rate and produces low numbers of false-positive results. At the time of publica-

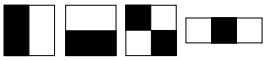

Figure 2.1: Various rectangle features

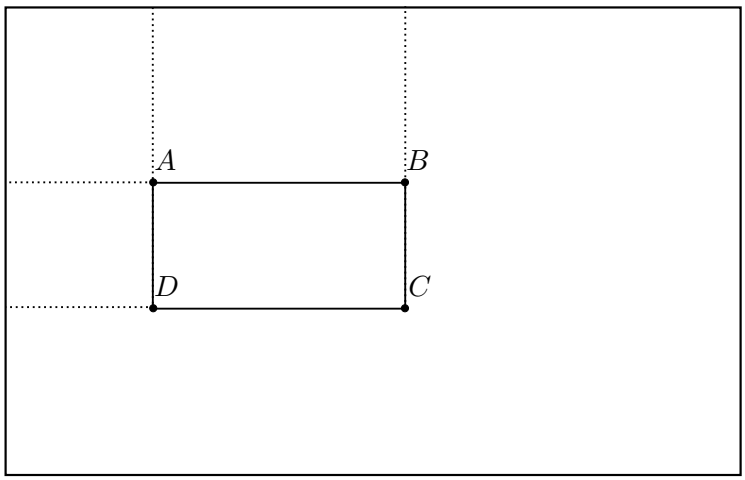

Figure 2.2: Demonstration of the Integral Image approach.

tion, it was significantly faster than previously used algorithms, the speed of the algorithm made it possible to be run in real-time at 15 frames per second.

The first concept that was used were Haar features shown in figure 2.1. These are rectangular features which are used to compute differences of the sums of pixels in an area inside the area of the rectangle. The sum of the pixels which lie within the white pixels are subtracted from the pixels which lie within the black areas, this calculation provides a score. Each Haar feature can be thought of as a weak classifier, the Viola–Jones algorithm trains many of these rectangular features using an ensemble training algorithm similar to AdaBoost. This training algorithm trains several weak classifiers and combines their outputs in a weighted sum, which represents the output of the ensemble classifier. If AdaBoost finds that a certain feature consistently predicts the correct result, its weight in the sum is increased or "boosted". Since many of the Haar features resemble different parts of the face, using this classifier leads to a considerably high accuracy.

The second is the Integral Image representation which allows the algorithm to extract the features very quickly. The integral image representation allows for much faster computation of the scores for individual Haar features. This representation turns the image into a summed-area table of the same size as the original image. The value of each pixel is replaced by the value of the summed pixels in the rectangle to the left and above relative from the pixel location. This enables the computation of the sum of any rectangular area within the image using only four lookups as demonstrated in the figure 2.2. In the provided example, the area of the rectangle between points A, B, C and D can be computed using the following formula.

$$\text{area} = I(C) + I(A) - I(B) - I(D) \tag{2.1}$$

where $I(A)$ is the value of the integral image for point A.

Histogram of oriented gradients This method was originally used for pedestrian detection in 2005 by Dala and Triggs [7]. This method uses a set of overlapping histogram of oriented gradients descriptors in conjunction with a support vector machine. This algorithm uses a sliding window of constant size passing over the input image to find regions of interest. In case there are multiple overlapping regions with a positive detection response, these can then be merged.

The following algorithm is ran for each window position. For each pixel, horizontal and vertical gradients are calculated by passing 1D filters $[-1, 0, 1]$ and $[-1, 0, 1]^T$ respectively, in this case these gradients represent the change of pixel intensity. For each such position, gradient magnitude and gradient angle are calculated. The image is then separated into cells of 8×8 pixels. These cells are then processed in overlapping 16×16 blocks of 4 cells, within which the gradients are grouped into nine bins of 20 degrees, these are the histograms of gradients. A support vector machine classifying whether or not the current window contains the searched object is then trained on this representation.

At the time of release of this paper, the histogram of oriented gradients method significantly outperformed all previously used algorithms.

Neural network based methods Approaches to face detection based on neural networks are often based on more general object recognition networks. The input of these neural networks are generally subregions of the input image. The image is partitioned into subregions, these subregions are then scaled to a constant size and passed to a convolutional neural network. The neural network then classifies these regions and outputs a prediction of what class the object in the region belongs to. In the case of face detection networks this prediction is a binary classification of whether there is a face present in the given region. A notable example of an object detection network is the YOLO network introduced in 2015 [35]. This network divides the input image into a grid of cells, these cells are then evaluated and a prediction is then made as to whether the subregion contains an object or not. The cell into which the center of an object falls is then responsible for detecting the object. The cells then classify the object which they contain. These classifications are then merged and outputs are produced. The outputs consist of a bounding box encompassing the object, a confidence score of whether the box contains an object and a confidence score of whether the class of the object is predicted correctly. A YOLO network can be used for face detection since a face detection task is essentially an object detection task with a single class. Other kinds of networks are also used for face detection, some convolutional networks simply transform an input photo into a heatmap representing the likelihood of a face being in a given location in the input image.

2.1.2 Face recognition methods

After the face has been located a face recognition algorithm can be used to try to find a matching face in its dataset, before this step some transformations are generally applied to try to improve the accuracy. This is a challenging task since the appearance of a person's face on a photo can be significantly altered depending on various circumstances and with this the performance of face recognition is also changed.

Factors affecting face recognition accuracy The factors which influence accuracy of face recognition can be divided into two groups: intrinsic and extrinsic factors. This section is based on a paper focusing on these factors by Anwarul and Dahiya [2].

The intrinsic factors are the results of natural processes of the body, such as hair and facial hair growth and unnatural factors such as plastic surgery. These factors can not be eliminated through the use of preprocessing and the facial recognition algorithms have to be robust enough to still perform well.

Extrinsic factors can typically be adjusted using preprocessing techniques. These factors include the following:

- Low resolution – Surveillance cameras typically capture a wide area, therefore the faces captured by these cameras will typically have low resolution making the use of facial recognition on such images difficult. This issue can be partially alleviated through the use of super-resolution neural networks.
- Noise – Digital cameras are prone to producing noisy images, this noise can be compensated by using a denoising filter during preprocessing.
- Illumination – Uneven lighting can obscure some areas of a face on the unlit side and make feature detection on the illuminated side hard or impossible.
- Pose variation – If the person pictured in the image is not facing the camera directly, facial recognition systems may perform poorly. To combat this, techniques which map the face on a 3D model with the use of facial landmarks have been developed.
- Occlusion – Faces may be partially hidden by hair, sunglasses or scarves. Occlusion makes the task of face recognition significantly harder. There exist methods which attempt to create recognition algorithms robust enough that they would not be as affected. In this thesis an approach which tries to reconstruct these occluded areas will be presented.

There are generally two approaches to face recognition, holistic methods and local methods.

Local methods Local methods are based on the usage of facial landmarks. These landmarks are significant points on the face such as corners of eyes, pupils, nose and lips. This approach was originally developed in the 1970s. These older approaches were based on finding these landmark points, measuring the distances between them and using ratios as representations of a face [21]. These representations could then be used to try to find the nearest neighbour, possibly producing a match. This approach has evolved towards the current state where landmarks are used for transformations of the facial imagery, which are then used as inputs of convolutional neural networks, which will be discussed later.

Holistic methods Holistic methods forego the extraction of facial landmarks, instead focusing on faces as a whole. The holistic methods were introduced in 1986 by Sirovich and Kirby with the use of eigenfaces [38]. This method is based on a representation of faces in a so-called face space. The premise behind face space is that an image of a face can be compressed and reconstructed by starting with a mean face image adding a small number of signed images. These images, as seen in figure 2.3, were created by using principal component analysis on a dataset of training images. A picture of a face could then be reconstructed back through adding a weighted sum of these eigenfaces to a base image.

A method for face recognition based on eigenfaces was proposed in 1991 by Turk and Pentland [42]. This method works by projecting the query image using PCA into the

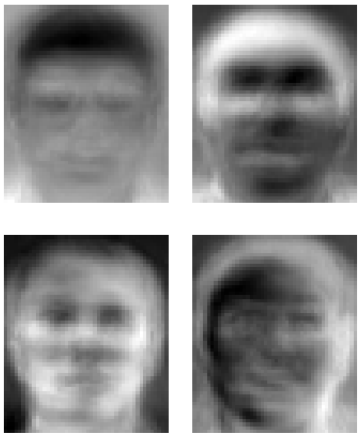

Figure 2.3: Eigenface representations. Image from AT&T Laboratories Cambridge [3].

eigenspace and searching for the nearest neighbour in the face space. In this paper, Turk and Pentland also showed that this method can be used for face reconstruction, they however only provided one sample which was also a part of the training set.

Neural network based methods Approaches to face recognition with the use of neural networks have had much success in recent years. A notable example of a face recognition network is the one used in the DeepFace approach developed by Facebook [40]. The DeepFace system first detects a face in the input image, this image is then aligned. The aligned face is then mapped onto a 3D face model which is then rotated so that the face is facing the camera directly. This step which the authors call frontalization compensates for pose variation. The frontalized face is then passed to a neural network which outputs a vector of values representing the identity of the person in the photo. This vector can then be assigned to a specific identity by finding the closest match in a database of representations using a weighted distance function. The authors of the paper claim to have achieved an accuracy of 97% which is very close to human-level of accuracy which is at 97.5%.

2.2 Neural networks

In the present, neural networks are the dominant approach towards machine learning. The basis for the approach was laid in 1958 with the introduction of the Perceptron, an artificial model of a simplified biological neuron. This artificial neuron was capable of learning linearly separable patterns and did not see much use in the future. It was found that for a neural network to learn non-linearly separable patterns, more layers would be needed. This however, was impractical due to the workings of the Perceptron learning algorithm. Further research into neural networks was stifled by the lack of needed processing power and only started to progress in the 1980s with the introduction of the backpropagation algorithm and increase in processing power. Since this time neural networks have made massive progress and are one of the most rapidly progressing areas in computer science. Neural networks are nowadays commonly used for a variety of tasks including classification, regression, image processing and generation and others.

The current basic model of a neuron maps a vector of its inputs to a single output value. The neuron contains a representation of its knowledge in a so-called vector of weights. The

output value is produced using a function that produces a single value from the combination of the input vector and the weight vector. This value is further processed using the activation function which produces the final output of the artificial neuron.

2.2.1 Activation functions

Activation functions are used to transform the output of a neuron, usually in a non-linear way. After this transformation the output can be used as an input for the next layer. If the chosen activation function is non-linear, then the neural network can approximate non-linear functions, otherwise it is equivalent to a single-layer neural network. There are various functions which are used for this purpose and the choice of activation function can drastically improve the convergence time of the network.

Binary step The binary step function was used in the original Perceptron model. This activation function provides a binary output and is therefore not suitable for the creation of multi-layered neural networks. Since it is also not differentiable, it can not be used in modern network training algorithms such as Stochastic gradient descent. The binary step function is as follows.

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{2.2}$$

where x is the input value.

Linear Like the binary step function, the linear activation function is not widely used. The first issue with this function is that, as mentioned earlier, it does not introduce any non-linearity into the neural network and any multi-layer neural network using this activation function has the power of a single-layer network. The second issue with this function is that the derivative is constant and does not depend on the input, as a result the function can not be used with Stochastic gradient descent either.

$$f(x) = x \tag{2.3}$$

where x is the input value.

Logistic sigmoid The logistic sigmoid function is an activation function which maps its inputs into the range $(0, 1)$. An advantage of this function is that it clamps results to this range. A major downside is that this function is prone to the vanishing gradient problem. Another downside is that this function is not centred at zero, which leads to slower convergence [27]. It is also fairly computationally expensive. This function is used in many applications apart from neural networks, including sociology, statistics, economics and others.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

where x is the input value.

Hyperbolic tangent The hyperbolic tangent function maps its inputs into the range $(-1, 1)$. It is a better choice than the logistic sigmoid function, since it converges faster [27]. This function otherwise suffers the same issues as the sigmoid function.

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

where x is the input value.

ReLU and its variants The use of the Rectified Linear Unit as an activation function which has been shown to perform significantly better than hyperbolic tangent or logistic sigmoid [9]. This function is currently one of the most commonly used functions in deep neural networks and convolutional neural networks.

ReLUs suffer from so-called Dying ReLU problem an issue similar to vanishing gradients, where the neurons "die", they produce the output of zero regardless of the input. This is caused by the neuron consistently producing a value smaller than zero, which gets evaluated by the ReLU to zero. Recovering from this state is unlikely, since the value of the derivative for $x < 0$ is constant at zero. This issue will be described in further detail later in this thesis.

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.6}$$

where x is the input value.

The success of ReLU has spawned several similar approaches which attempt to solve the dying ReLU problem by either modifying the slope of the function for $x < 0$ or replacing it with a non-linear curve. Notable examples are Leaky ReLU and SELU.

Leaky ReLU This approach attempts to solve the dying ReLU problem by assigning a small positive slope to the function for $x < 0$ [31].

$$f(x) = \begin{cases} 0.01x & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.7}$$

where x is the input value.

Another modification of this approach is the the Parametric Rectified Linear Unit, or PReLU, in which the coefficient is learned instead of being set to a constant value as seen in the equation above [14]. Both of these approaches have been shown to perform better than conventional ReLU [45].

SELU Scaled Exponential Linear Units are an approach which completely eliminates the problems of vanishing and exploding gradients through allowing self-normalization within the network, resulting in overall better performance when compared to ReLU and variants of Leaky ReLU [24]. The equation is as follows.

$$f(x) = \lambda \begin{cases} \alpha(exp(x) - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{2.8}$$

where x is the input value and λ and α are the scaling parameters values of which were provided by the authors of the cited paper as $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$.

Swish Swish is a fairly new development in the area of activation functions, having been published in 2017 by researchers at Google. Swish is similar to the ReLU activation function, however it has been shown to have better performance in deep neural networks where the only change was replacing ReLU, or its variant, with Swish [34]. The major difference from ReLU is that Swish is smooth and non-monotonous. These factors may play a role in its performance.

$$f(x) = x \cdot \sigma(x) \tag{2.9}$$

where x is the input value and $\sigma(x)$ is the sigmoid function shown in 2.4

Softmax The Softmax activation function is mainly used in classification tasks where an input is supposed to be assigned to one of n classes. This function ensures that all outputs sum to one and the results can then be interpreted as the probability of a given sample being a member of given class.

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{2.10}$$

where $\vec{x}$ is the input vector and J is the length of the input vector.

2.2.2 Loss functions

This section is based on [10] and [4]. An objective function is a function which we seek to minimize or maximize during the training of the neural network. A loss function, also known as error function, is a special case of an objective function which we seek to minimize. Loss functions used in neural networks generally somehow show the difference between the output of the neural network and the desired output. The task of training algorithms is to find the weights which result in minimizing the loss function for a given input.

Mean absolute error The mean absolute error loss, also L1 loss, specifies the mean absolute difference between the actual output of the neural network and the target values. This loss function is most commonly used in regression models.

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |y_i - t_i| \tag{2.11}$$

where y is the vector of predicted values and t is the target vector.

Mean squared error The mean squared error loss, or L2 loss, computes the mean squared difference between the results produced by the neural network and the target values. Like mean absolute error, this loss is often used in regression models.

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - t_i)^2 \tag{2.12}$$

where y is the vector of predicted values and t is the target vector.

Cross entropy loss As opposed to the aforementioned functions, the cross entropy loss is used primarily in classification tasks in which each input is assigned into one of K mutually exclusive classes [4]. If used for classification, this loss function is to be used with the softmax activation function.

$$CE = -\sum_{i=1}^{K} t_k \ln y_k \tag{2.13}$$

where t is the target class in one of K encoding and y is a vector where an item y_k is the predicted probability that a given sample belongs to class k.

2.2.3 Layers

This section is based on [10] and [4]. The commonly used neural networks consist of stacked layers which transform inputs to outputs in various ways. In this section, an overview of the most frequently used layers will be given.

Fully connected layer The fully connected layer is the basic building block of neural networks. Every neuron in this layer is connected to all of the neurons of the previous layer. This means that the number of parameters needed increases rapidly with the number of neurons in each layer. With the increase in complexity of the tasks for the solution of which neural networks are used, it has become apparent that the number of computations needed for example for the evaluation of images is too high and therefore convolutional layers were introduced.

Convolutional layer The convolutional layers were introduced to alleviate the problem of the high number of parameters of fully connected layers and enabled the training of deep neural networks for image and natural language processing.

These layers are based on the operation of convolution which is described by the following function. The convolution operation is typically denoted using a $*$ symbol.

$$s(t) = (x * w)(t) = \int x(a)w(t-a)\mathrm{d}a \tag{2.14}$$

where $x(t)$ is the input and $w(t)$ is a weighting function, in the area of convolutions it is commonly known as kernel. The output is sometimes also referred to as the feature map. In practice however, neural networks commonly work with discretized data, we can assume that the function will only be defined for an integer t, then the discrete convolution function is defined like so:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.15}$$

In convolutional neural networks the input will commonly be a multi-dimensional array called a tensor. For example an RGB image of 300×300 pixels will be a tensor of dimensions $300 \times 300 \times 3$, since each colour will be in a separate dimension, this third dimension is called a channel. For real-world application both the input tensor and the kernel will typically be finite, so we can simplify the equation for convolution even further to the following.

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i-m, j-n)K(m,n) \tag{2.16}$$

where $S(i, j)$ is the value of convolution for the position i, j, K is the kernel and I is the input image.

In the context of convolutional layers, the kernel serves the same purpose as the weight vector in fully connected layers. The kernel is learned over the course of training. The main advantage of using convolutional layer instead of fully connected layers is that the weights of a kernel are shared for all inputs, this means that there is a significantly lower number of parameters to optimize, this results in much lower computation times and training times. Another advantage of convolutional networks in image processing is that they are not vulnerable to shifting. This means that when the input is shifted by one pixel, then the output of the layer will also only be shifted by one pixel, this is not the case in fully connected layers which would produce a vastly different output. The area covered by a kernel is referred to as the receptive field of the convolution. The larger the kernel, the more contextual information is available for the convolutional layer, but increasing the size of the kernel also means increasing the number of parameters and so increasing the computation time.

Convolutional layers typically have several parameters:

- Kernel size – This simply denotes the height and width of the kernel.
- Padding – One property of convolution is that for kernel size higher than 1 it produces an output which is smaller than the input because by the edges of the image, the kernel would need to use data which is out of bounds of the image. This is typically desirable, since it allows the convolutional layer to distil useful higher level features into a smaller output. In the cases where the output size should not be lowered, the input image can be zero-padded to allow the output be the same size as the input.
- Stride – The stride of convolution selects by how many pixels the kernel moves in each step. This parameter is used when it is desirable to decrease the output tensor dimensions,
- Dilation – Dilation specifies the size of gaps between the items of a kernel. Dilated convolutions are used as a means of increasing the receptive field without increasing the number of parameters [47]. The use of dilated convolutions serves as an effective means of gathering context from neighbouring areas of the central pixel. A conventional convolution has a dilation of 1, for example a convolution with a kernel size of 3×3 and dilation of 1 will have a receptive field of size 3×3. A convolution with kernel size of 3×3 and dilation of 2 will have a receptive field of 7×7, but the number of parameters will stay the same.

The size of the dimensions of the output with the relation to the input and parameters can then be calculated with the following formula:

$$o = \left\lfloor \frac{i + 2p - d * (k - 1) - 1}{s} \right\rfloor \tag{2.17}$$

where o is the output size, i is the input size, d is the dilation, k is the kernel size, p is the amount of padding and s is the stride [41].

Pooling layer Pooling layers are an important part of convolutional neural networks, the typical structure is that a convolutional layer is followed by a non-linear activation function and that is then followed by the pooling layer.

It works similarly to kernel layers in that there is a window which is gradually moved over the input data, however in the case of pooling layers the output is a summary statistic of the input. There are many different types of pooling layers, the most commonly used ones are the max-pooling layer and the average pooling-layer. The max-pooling layer selects the maximum value in a certain neighbourhood and reports this value. The average-pooling layer computes the average value and reports it.

The pooling layer serves three main purposes, first of all it reduces the dimensions of its input, effectively working as a downsampling layer. The second purpose is reducing noise in the processed data. The third purpose is that it makes convolutional neural networks approximately invariant to shifts in the input.

Batch normalization layer The batch normalization layer is a normalization mechanism which significantly speeds up training of neural networks [17]. The goal of batch normalization is to achieve a stable distribution of activation values, since the distributions of each layers change as the preceding layers change parameters. This layer adds two trainable parameters, mean and standard deviation, using these parameters the layer normalizes its inputs by subtracting the mean and dividing by the standard deviation. This allows for use of higher learning rates and because it also acts as a regularizer it makes the network less prone to overfitting.

Transposed convolutional layer This layer is commonly used in situations where an image is supposed to be upscaled, or generated. The transposed convolutional layer serves the purpose of doing the convolution operation in the opposite direction from ordinary convolution. While a convolutional layer would decrease the dimensions of the input, the transposed convolutional layer serves to increase the dimensions of the input. This layer can for example be used in a generator network of a GAN, or the decoding part of an autoencoder [8].

2.2.4 Convolutional neural networks

Convolutional neural networks are a family of deep neural networks which use convolutional layers. Fully connected layers are typically used in the last layers to transform the features extracted by the convolutional layers into the desired representation. These networks are generally used to extract data from images, or to transform the images.

The AlexNet convolutional neural network was introduced in 2012 by Alex Krizhevsky [25]. The authors utilized hardware accelerated training to train a deep convolutional network containing eight layers, the first five of which were convolutional layers, some of which were followed by max-pooling layers, and three fully connected layers. The use of hardware acceleration allowed the authors to train a much deeper model than was common. The depth of the model allowed AlexNet to far exceed all architectures which were used before.

Attempting to further increase the depth of CNNs was stifled by the issue of vanishing gradients, preventing the architectures from becoming deeper. This issue was resolved in 2015 with the publication of ResNet by He et al. [13]. The ResNet architecture presented a new approach towards CNN architectures with the inclusion of residual connections. These connections skip one or more layers and bring unchanged data further into the network, thus allowing the gradient to be propagated back through the network. In the ResNet paper, the authors managed to increase the number of trainable layers to up to 1000 layers,

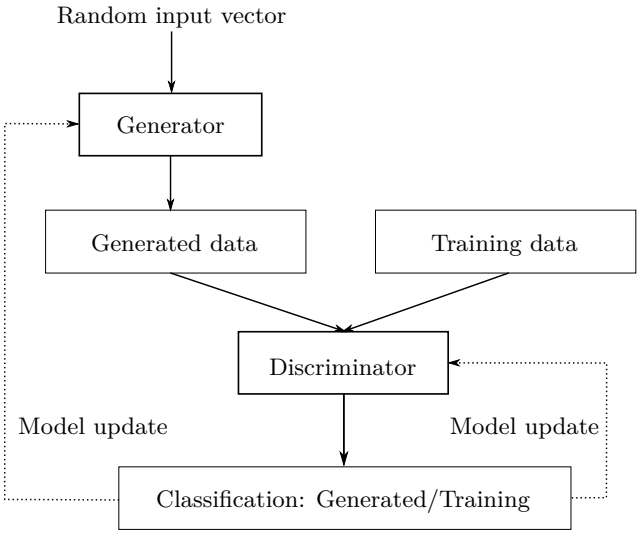

Figure 2.4: A diagram of the GAN model architecture. Adapted from [5].

this was a massive improvement over the previously state of the art VGG networks which had a maximal depth of 19 layers [37].

2.2.5 Generative adversarial networks

Generative adversarial networks, or GANs, are a generative model which was introduced in 2014 by Goodfellow et al. [11], this section is based on the same paper. Before the release of GANs deep learning models were mostly successful in tasks with high-dimensional inputs and lower dimensional outputs, such classification, or object detection. Generative models were generally not as successful until the introduction of GANs.

This network architecture is based on two competing neural networks, the discriminator and the generator as shown in figure 2.4. These two networks compete against each other. The authors of the original paper liken the generative model to currency counterfeiters who try to produce fake currency and successfully use it and the discriminator to a team of anti-counterfeiting police officers who try to detect the fake money. Eventually this cat and mouse game will lead to the counterfeiters producing fake money which is indistinguishable from genuine currency. Like in the analogy, the generator network produces samples which should be as close to the training set as possible, while the discriminator network tries to distinguish between genuine training set samples and fake samples created by the generator.

The generator tries to learn the distribution p_g over data $\boldsymbol{x}$, for this a distribution of random noise input variables $p_{\boldsymbol{z}}(\boldsymbol{z})$ is defined, the generator then represents a mapping $G(\boldsymbol{z};\theta_g)$ into the data space where G is a differentiable function represented by a neural network with parameters θ_g. The discriminator is represented by $D(\boldsymbol{x},\theta_d)$ which outputs $D(\boldsymbol{x})$, the probability that the data is from the training set $\boldsymbol{x}$ rather than generated data from the discriminator. The discriminator is trained to maximize the probability of correctly guessing whether the data was generated, or comes from the training dataset. At the same time the generator is trained to minimize $log(1 - D(G(z)))$. These two networks end up playing a minimax game with the function $V(G, D)$.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{2.18}$$

Early in training, the discriminator can overtake the generator and reject all generated samples with high confidence, in this case the generator can be trained to maximize $\log(D(G(\boldsymbol{z})))$ instead of minimizing $log(1-D(G(z)))$. This will provide a stronger generator in the beginning of training.

The success of GANs has spawned many similar architectures which tackle different generation tasks. The deep convolutional generative adversarial network was proposed in 2015 by Radford et al. This architecture replaced the fully connected layers of the original GAN with convolutional layers and as a result was the first to be successfully able to generate high resolution face images [33].

2.3 Neural network training

Neural network training is an optimization task where the weights are iteratively modified to optimize a loss function, this section will focus on methods which are commonly used for training. Neural network training is not guaranteed to be successful and there are several problems which can halt the training of the network, or can render the network unusable. The last part of this section is about these issues and provides strategies on how to avoid them.

2.3.1 Stochastic Gradient Descent

This section is based on [4]. As stated earlier, neural networks are one approach to solving an optimization problem. The task of a neural network is to find weights which minimize the loss for a set of inputs and outputs, a so-called training set. One approach to minimizing the loss is using gradient descent, a neural network training algorithm.

Gradient descent is an iterative optimization algorithm which utilizes partial derivatives to find the gradient of the loss function. Since the objective is to minimize the loss function and the error function increases in the direction of the gradient, the gradient descent algorithm iteratively updates the weights of the neural network to move against the direction of the gradient. The algorithm runs in iterations until the loss function reaches a point at which it stops decreasing, or for a set number of iterations.

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E(\mathbf{w}^{\tau}) \tag{2.19}$$

where $\mathbf{w}^{\tau}$ is the weight vector in a step τ, E is the error function and $\eta > 0$ is a parameter known as learning rate.

After each update the neural network has to be re-evaluated, so that a new value of the error function can be computed.

Gradient descent works in batches, which means that for one step to be computed, the entire training set needs to be evaluated. This has shown as prohibitively slow when using large datasets and led to long training times before convergence. The solution to this issue is the usage of stochastic gradient descent. SGD is an on-line algorithm which updates the weight vector with each data point. Stochastic gradient descent has been shown to produce significantly superior results in less time [26]. Stochastic gradient descent can either go through the dataset sequentially, or in a random order. The benefit of using SGD is that the short update intervals add noise into the training process, making the algorithm less likely to get stuck in a local optimum. Another advantage is that on-line methods handle redundancy much better than batch methods.

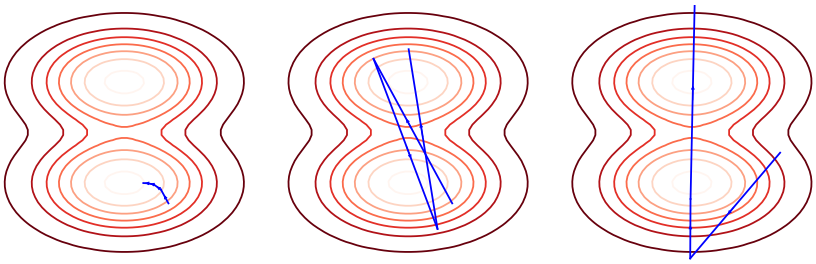

Figure 2.5: Learning rate selection. From left to right: 1) learning rate is small enough to converge around a minimum, 2) moderate so that it bounces among minima, 3) too large to converge. Image from [46].

Learning rate Setting the learning rate parameter to a right value is crucial. Setting the learning late too low will make the algorithm slower to converge and might make it more likely to get stuck in a local minimum, while setting it too high might make SGD overshoot the global minimum and run for longer than needed as seen in 2.5. Learning rate decay is a technique used in neural network training, where the learning rate is set to a high value at the start of training and over the course of iteration, the learning rate is decreased either gradually or several times in steps. The theory behind this approach is that the using the higher learning rate at the start of training will make the algorithm reach the proximity of the global minimum faster and after getting into the proximity of the global minimum the decreased learning rate will make it avoid oscillating around the optimum and eventually allow it converge. Another possible explanation for why learning rate decay increases the efficiency of training is that the initial high learning rate prevents the network from precisely learning noisy data, helping it generalize, while the decayed learning rate enables the network to learn complex data patterns [46].

2.3.2 Adam

This section is based on the original Adam paper [23]. The Adam algorithm is an optimization algorithm which utilizes momentum, the name of the algorithm comes from "adaptive moment estimation". Like stochastic gradient descent, Adam is an on-line optimization algorithm which iteratively updates the weights of the network. This algorithm however is an adaptive algorithm, which means that it computes individual learning rates for individual parameters and updates them during training. Adam achieves this by keeping exponential moving averages of the gradient and the squared gradient, the momentum of the gradient. These values are updated each step and after bias correction they are used in the update rule instead of the gradient itself, as seen in formula 2.20. Through the use of momentum, Adam is less likely to get stuck in local minima and passes through saddle points, since the moving averages act as a "smoothing" factor allowing the optimizer to pass through such points quickly.

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \frac{\hat{m}^{\tau}}{\sqrt{\hat{v}^{\tau}} + \epsilon} \tag{2.20}$$

where $\mathbf{w}$ is the weight vector, η is the step size, $\hat{m}$ is the bias-corrected first moment estimate (the moving average of the gradient) and $\hat{v}$ is the bias-corrected second raw moment estimate (the moving average of the square of the gradient)

2.3.3 Backpropagation

This section is based on chapter 2 of [10]. The backpropagation algorithm enables the training of multilayer feed-forward neural networks through distribution of the network error back through the layers and adjusting their weight accordingly. It is by itself not a neural network training algorithm, it is rather a mechanism that allows the information about the loss to be propagated back through the network in order to compute the gradient. For the network to be trained, another algorithm has to be used, such as Stochastic Gradient Descent. Backpropagation utilizes recursive application of chain rule to distribute the gradient to all layers of the neural network.

2.3.4 Issues in neural network training

Over the course of training of a neural network, several issues can occur. It is vital to have knowledge of what those are and how to prevent them. This section is based on [10].

Vanishing gradients The vanishing gradient problem occurs primarily with the use of sigmoid and hyperbolic tangent activation function. This problem occurs in these functions because they map the entire input range into a very small range on the output. For example for input values far from zero in the negative direction, even when a large change in input values occurs, the value of the gradient will be very small. This can cause the neural network to significantly slow down the training speed, in extreme cases the network can completely lose the ability to learn.

A similar issue can also happen when the gradient approaches a sharp "cliff" in the error function. In this case the optimization algorithm will dramatically overshoot the optimum.

This paragraph is based on [13]. Traditionally, these issues were mitigated through the use of normalized initialization of weights and intermediate normalization layers, however these measures were not effective in very deep neural network models where adding more layers would result in higher training errors. Creating deeper neural networks was enabled with the introduction of residual networks. Residual networks add skip connections which skip one or more layers and bring unchanged data further through the network. Through these connections gradients can be backpropagated with less risk of vanishing gradients.

Dying ReLU The vanishing gradient issue can also be mitigated through the use of different activation functions, such as ReLU, this activation function can however suffer from the dying ReLU problem.

This paragraph is based on [30]. The dying ReLU is a problem that is related to the vanishing gradient problem, when a ReLU neuron dies, it produces an output of zero for any input. When this occurs in enough neurons in the network the network dies, it also produces an output of zero for any input. This issue can be mitigated or completely overcome through the use of several methods. The authors of the cited paper propose a method of initialization which mitigates the problem. Other methods of avoiding ReLU death include the use of batch normalization layers, which also mitigate the problem, however at a cost of 30% of computational overhead. Another approach to eliminating ReLU death is the use of a different activation function, such as SELU or Swish.

Overfitting When training neural networks, the neural network is generally trained using a training dataset, while its performance is evaluated by its accuracy on testing data.

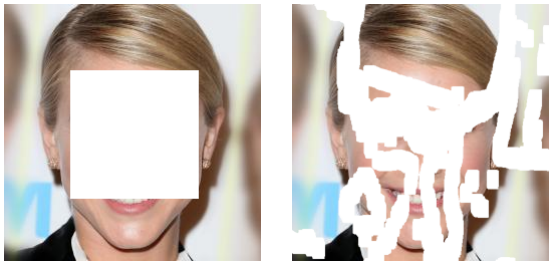

Figure 2.6: A comparison of regular (left) and irregular (right) hole masks applied to an image.

Overfitting is an issue that occurs when the neural networks learns the noisiness of the training dataset too precisely, this leads to the loss of ability to generalize on the testing data. The neural network essentially learns a function which becomes too focused on the outliers in the training data.

This paragraph is based on [32]. Overfitting can be combated through the use of early-stopping. When a neural network begins to overfit, the training error continues to decrease, while testing error will start to rise. The difference between the training error and the testing error is called the generalization gap. The algorithm usually uses a mechanism of quantifying this gap to a more useful metric through the use of a loss function, or simply stops training after the generalization gap increases in n successive steps. Early-stopping detects the start of overfitting and stops training before the neural network learns to overfit.

Mode collapse This paragraph is based on [1]. The task of generative adversarial networks is to generate new samples of data which resemble the training set. Mode collapse in GANs is a problem that occurs during training when the generator network starts to produce the same sample or a small group of samples in every run regardless of the input of the network. Another variant of this problem is mode missing, which is characterized not by consistently generating the same sample, rather some types of samples are missing in the generated data. This issue can be alleviated by modifying the structure of the generator network, or using different optimization algorithms.

2.4 Face reconstruction

Face reconstruction is the main topic of this thesis, in most sources it is also referred to as face inpainting, these terms will be used interchangeably throughout this text. The task comes down to accurately repairing an image of a face where a part has been masked or obscured. More broadly speaking a neural network performing image inpainting should fill missing data with data it deems to be probable based on the unmasked part of the image and previously learned data. Image reconstruction tasks used to be mainly focused on regular hole masks, typically a 64×64 square cut-out mask in the centre of a 128×128 pixel image as shown in figure 2.6. This type of masking could however not realistically be used in practice. Because of this, recently there has been a move towards free-form inpainting tasks, which use irregular masks which can be located anywhere within the image. The models trained for irregular mask inpainting are able to generalize better and are more suitable for practical use. For this reason, this thesis will focus solely on free-form face inpainting.

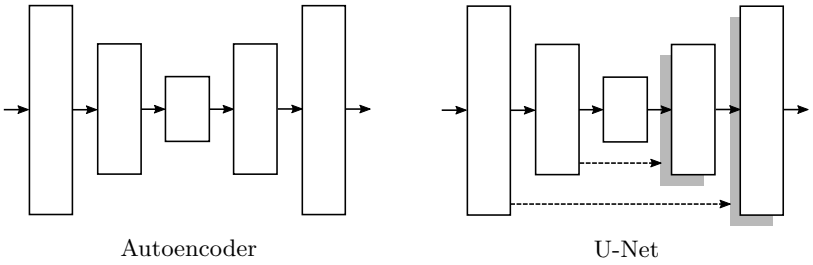

Figure 2.7: A comparison of network architectures of autoencoders and U-Nets. Adapted from [18].

Networks for these tasks are in present most commonly GANs with the generator being based on an autoencoder-like architecture. This section covers techniques commonly used in the creation of neural networks for face reconstruction.

2.4.1 Autoencoders

Autoencoders are an architecture of feedforward neural networks which was developed mainly for representation learning tasks, the objective of these tasks is to find a smaller representation of the input data, while being able to reconstruct the data back to its original size as closely as possible. During the course of training, these networks use the same data as both the input and the output. A network of the autoencoder architecture is composed of two subnetworks, the encoder and the decoder, with a bottleneck in between. In the image processing context a convolutional autoencoder is used, the encoder is composed of convolutional layers which reduce the dimensionality of the input data and the decoder uses upsampling, or transposed convolutions to increase the dimensionality of the data back to its original size. The bottleneck region constricts the amount of information that flows through the network and prevents the autoencoder from simply learning an identity function. Because the network is forced to pay more attention to the features of the input which distinguish it from other datapoints in the training set the most, the representation often contains data which contains useful properties of the training dataset.

Rather than their original purpose, autoencoders were discovered to be useful in their ability to reconstruct damaged data and generate new samples, networks based on this idea are the denoising autoencoders. In the case of these networks, the data on which they are trained is different. A datapoint is sampled and random noise is added, this noisy datapoint is then used as the input of the network and the original clean data is used as the target. These networks lay foundation on the original two part structure of autoencoders, they however do not have the need to constrain the size of the data in the bottleneck area since the danger of the network learning an identity function is not present.

2.4.2 U-Net

The U-Net neural network was first proposed in 2015 for medical image segmentation [36]. It builds on the autoencoder network structure and modifies it by the addition of skip connections as shown in figure 2.7. These skip connections connect the layers which have the same dimensions and number of channels. The U-Net found its use in image transformation tasks because of its ability to propagate relatively unchanged data forward through the network via the skip connections and at the same time being able to modify

data in the inner layers. A good example of use of this type of network for image inpainting is the pix2pix network [18]. This network was used for different types of image translation tasks, including street photo image inpainting. While the pix2pix network was not used for face inpainting, it presented the use of the U-Net structure as beneficial for inpainting tasks. It is especially useful for face inpainting, since it allows the propagation of fine details of the unmasked regions of the image, which might be lost with a conventional encoder-decoder network, while the inner layers extract data from the image context to fill in the missing data.

2.4.3 Specialized layers

The task of image inpainting has a specific issue tied to the task itself stemming from the fact that a part of the image is obscured by a mask. This makes the use of conventional convolutional layers ill-fitted to the task, since a convolutional kernel is applied all over the input image, this will include the areas of the input which are completely obscured by the mask and thus do not contain any meaningful data for the network. This somewhat complicates the task and several different approaches were developed for the explicit purpose of eliminating the influence of the masked regions.

Partial convolutional layer This section is based on the original partial convolution paper [28]. The partial convolutional layer introduced in 2018 tries to address the issue of applying the convolutional kernel by passing a mask through the network which masks and renormalizes the output of the convolution. This mask is also updated after each layer, it is progressively getting smaller and after a number of update steps it disappears completely, making the layer act effectively as a normal convolutional layer.

Let $\mathbf{W}$ be the convolution filter weights and b be the bias of the convolution, $\mathbf{X}$ are the pixel values covered by the kernel, and $\mathbf{M}$ the corresponding binary mask. The partial convolution is then defined as follows:

$$x' = \begin{cases} \mathbf{W}^T(\mathbf{X} \odot \mathbf{M}\frac{\text{sum}(\mathbf{1})}{\text{sum}(\mathbf{M})} + b) & \text{for sum}(\mathbf{M}) > 0 \\ 0 & \text{else} \end{cases} \tag{2.21}$$

where $\odot$ is element-wise multiplication, $\mathbf{1}$ is a ones matrix of the same shape as $\mathbf{M}$. The element-wise multiplication causes the convolution to only take the unmasked pixels into account, this is further balanced by the scaling term $\frac{\text{sum}(\mathbf{1})}{\text{sum}(\mathbf{M})}$ which compensates for the missing masked pixels.

After each partial convolution, the masks are updated with the following update rule:

$$m' = \begin{cases} 1 & \text{for sum}(\mathbf{M}) > 0 \\ 0 & \text{else} \end{cases} \tag{2.22}$$

where m' is the updated mask pixel for a given kernel location.

As can be seen, the update rule will cause the holes in masks to gradually shrink until the whole mask is just the values of 1. The benefit of this approach is that it significantly boosts performance in inpainting tasks at a relatively low computational cost. The disadvantage is that the mask has to be passed through the network along with the data, in cases where the dimensions of the passed data are increased, the mask has to be upsampled. Also when skip connections using concatenation are used, the two groups masks of merging data have to be concatenated as well. In the case of skip connections using element-wise addition, the

masks have to be merged using the "or" function. The performance of partial convolutional layers was later surpassed by the gated convolutional layer.

Gated convolutional layer This section is based on the paper which introduced gated convolutional layers in 2018 [48]. Gated convolution was designed to improve performance of partial convolutional layers by adding soft gating on a learnable mask. No mask is explicitly provided to this layer, it is instead inferred from the data, this proves beneficial for several reasons.

The first reason is that the masks are used all throughout the network, if partial convolutions were used, the mask would gradually disappear due to the mask update rule. This is not the case for gated convolutional layers, the learned mask allows the network to focus on the region where data should be inpainted throughout the whole network. The second reason is that, as shown by the original paper, in deeper layers the mask providing convolution learns to produce semantic segmentation and other masking outputs which further boost the performance of the network. Another advantage over partial convolutional layers is that gated convolutional layers learn one mask for each channel of data as opposed to one mask for all channels, this allows gated convolutions to learn to mask features specifically for certain channels.

A gated convolutional layer consists of two convolutional filters. The $Feature$ filter processes input data as is common with a conventional convolutional layer and the $Gating$ filter provides the learned mask. Gated convolutions are defined as follows:

$$Gating = \mathbf{W}_g \cdot \mathbf{X} \tag{2.23}$$

$$Feature = \mathbf{W}_f \cdot \mathbf{X} \tag{2.24}$$

$$x' = \phi(Feature) \odot \sigma(Gating) \tag{2.25}$$

where $\mathbf{W}_g$ are the convolution filter weights of the gating convolution, $\mathbf{W}_f$ are the filter weights of the feature extracting convolution, $\mathbf{X}$ are the pixel values covered by the kernels and x' is the output value of the convolution. The activation function ϕ can be chosen as any activation function and σ is the sigmoid function.

As can be seen, the usage of sigmoid function limits the gating activation to a range between zero and one, thus providing smooth gating as opposed to binary gating present in partial convolutional layers.

2.4.4 Dense multiscale fusion blocks

This section is based on the original dense multiscale fusion block (DMFB) paper [16] and the ESRGAN paper [44]. Before DMFBs can be introduced, an overview of the use of blocks in neural networks will be given.

Blocks as an architectural unit of computing within neural networks have been used since the introduction of residual blocks in ResNet which allowed for training of deeper models. Generally speaking, a block is an architectural unit consisting of several layers which serve to process data and a skip connection which propagates unchanged data through the block, these two streams of data are then combined by either element-wise addition, or concatenation to form the output of the block. The introduction of residual blocks allowed training of deeper models while avoiding the vanishing gradient problem. The first widely successful block was the dense block introduced in DenseNet, this block utilized multiple overlapping skip connections and achieved considerable success in classification

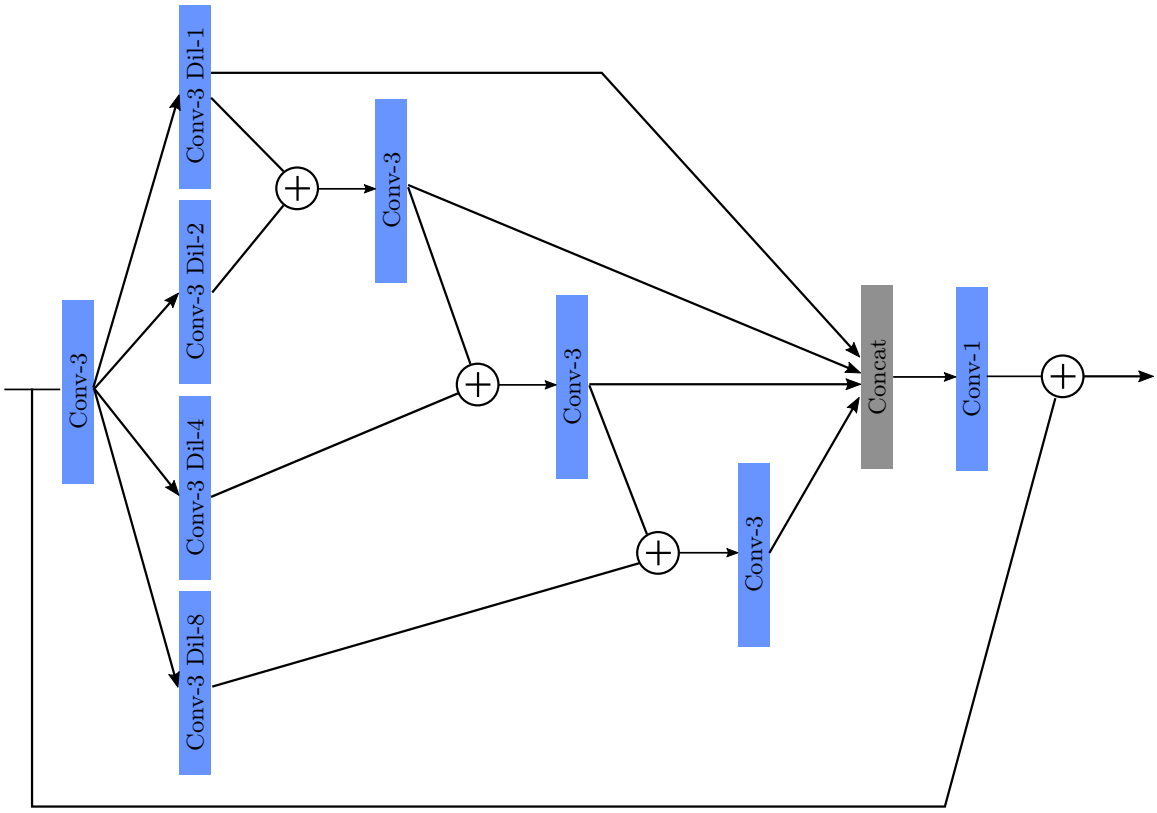

Figure 2.8: A diagram of the dense multiscale fusion block. "Conv-3 Dil-2" represents a convolutional layer with a kernel size of 3×3 and dilation of 2, the $\oplus$ sign represents element-wise addition.

tasks. The ESRGAN successfully used dense blocks with a GAN based network for image super-resolution, these blocks were however not successfully used in face inpainting tasks and seem to be more suited for other types of tasks.

The dense multiscale fusion blocks, inspired by dense blocks, were introduced specifically for face inpainting. These blocks contain several dilated convolutions with different dilation rates. Dilated convolutions have previously been shown to improve performance on inpainting tasks, since they are able to pull data from relatively distant pixels at a fairly low computational cost, especially when compared to convolutional layers with large kernels.

The structure of the DMFB is shown in figure 2.8. The block begins with a convolutional layer which reduces the number of channels to 64, which is then kept constant throughout the block. The main feature of the DMFB, which follows after the initial convolution, are four parallel dilated convolutions with dilation rates of 1, 2, 4 and 8 whose outputs are then gradually merged by other convolutional layers. These merged outputs are then concatenated and the number of channels is then adjusted by an addition convolution with a kernel size of 1. This block enhances the performance of a common dilated convolution and has fewer parameters than large kernels. All layers in the block are followed by instance normalization and ReLU activation.

2.4.5 Discriminators

In the field of face inpainting with the use of generative adversarial networks, several important developments were made, the two of most interest to this thesis are the PatchGAN and the use of spectral normalization. Both of these approaches are applied in the discriminators used by currently state-of-the-art approaches.

PatchGAN This section is based on the pix2pix paper [18]. The PatchGAN discriminator was introduced in 2016 in the pix2pix paper, this discriminator was used in conjunction with an U-Net like generator, as mentioned earlier. The idea behind this discriminator is that it does not simply give a single scalar output for the whole image as is usual with a basic discriminator network. The PatchGAN discriminator however gives a larger array of results, each of these results then refers to a subregion, or a patch, of the input image. In the original PatchGAN, the discriminator classified 70×70 pixel patches of the input imaged. This approach is equivalent to training an ensemble of discriminators taking overlapping 70×70 inputs. The output of the discriminator network is then a 3-D array of features, on which the conventional GAN loss is then applied. PatchGAN discriminators have later been successfully used in several face inpainting focused papers, such as the one which introduced gated convolutions, this paper utilized the PatchGAN in conjunction with spectral normalization.

Spectral normalization The following section is based on the gated convolution paper [48] and Spectral Normalization Explained, an article by Christian Cosgrove [6]. Spectral normalization is a technique used for improving the stability of discriminator network training. The weights of each layer in a spectral normalized network are subject to spectral normalization after each weight update step so that the spectral norm of the weight matrix, denoted by $\sigma(W)$ is equal to one. The normalization rule is the following:

$$W := W/\sigma(W) \tag{2.26}$$

where W is the weight matrix and $\sigma(W)$ is the spectral norm of the weight matrix. The spectral norm $\sigma(M)$ of a matrix M is the largest singular value of M, which is the square root of the largest eigenvalue of $M^T M$.

Spectral normalization is beneficial to neural network training in a similar manner to the commonly used batch normalization, in that it constrains the data to manageable ranges and thus stabilizes network training preventing gradient explosions and ReLU death. Furthermore, the use of spectral normalization reduces the risk of mode collapse when used in discriminator networks. Spectral normalization has also been found to improve training speed when compared to other normalization methods.

2.4.6 Losses

Modern generative adversarial networks commonly utilize multiple loss functions for training of the generator, as compared to the originally proposed GAN. This is almost necessary to prevent mode collapse of the network, since the original GAN was intended for unsupervised learning. Nowadays the use of combined loss functions is needed to constrain the generator network into only generating a specific range of cases for each input through the use of other loss functions while the adversarial loss serves to enforce that the generated outputs should be realistic. Most commonly for face inpainting tasks, the combined loss is

a combination of adversarial loss and L1 loss. In this case the L1 loss serves to produce low-frequency features and the adversarial loss produces high-frequency features. Other losses have however been developed specifically for image transformation and have found their use also in face inpainting, the most influential of them is the perceptual loss.

Perceptual loss This section is based on the perceptual loss paper [20]. This approach was presented in 2016 and was originally developed for style-transfer networks. The goal of a style-transfer network is to transform an input image so that its stylistic appearance is as similar to the target style as possible while preserving the structural look of the original image. A good example is transforming a photo of a landscape of a city so that it looks like a painting, the city should be recognisable, however the resulting image should look like a painting in the style of the target painting. The original paper achieved this by using a 16 layer VGG network pretrained on the ImageNet dataset as a loss network. This network is used to classify the style target and the activation maps of different layers within the loss network are then extracted. These extracted activation maps are then used as training targets for the network.

The approach used in the paper which introduced gated convolution claimed that usage of the perceptual loss in networks where spectrally normalized PatchGAN discriminators are used was not necessary, since the outputs for different patches already contain similar information needed for producing high-frequency features like the details of hair and others. Other approaches, such as the recent SC-FEGAN [19] successfully use perceptual losses in conjunction with SN-PatchGAN.

Chapter 3

Solution proposal and implementation

This chapter focuses on the implementation of the designed model, the choice of training datasets and tools which were used to produce a working model. As stated in the thesis assignment, the goal of this thesis is to design a model of a neural network for face reconstruction. Before a model can be designed, several questions as to the specifics of the task needed to be decided on. The following sections will present the choices that were made and the reasons behind those choices. The model which was designed is also presented in this chapter.

3.1 Datasets

The choice of a dataset significantly impacts both the task specification itself and also the maximum possible success rate of the final model. Choosing the right dataset is thus crucial for any machine learning task. There are face image datasets of varying quality and size. There are two main groups of face datasets, so called in-the-wild datasets and specially collected datasets. Specially collected datasets used to be more common in early days of machine learning, these datasets were often professionally shot in studios where lighting and pose of the subject being photographed could be controlled. These datasets were however usually quite small and did not offer enough diversity in terms of the number of subjects and their age, ethnicity and gender. The most common datasets are currently in-the-wild datasets. These datasets contain photos of people in imperfect conditions. The face could be facing any direction, the subject might be wearing sunglasses and head coverings and so on. Modern in-the-wild datasets generally contain photos of people of varying ages, ethnicities and genders. These datasets are also generally quite large, in tens of thousands of photos. Training models on in-the-wild datasets is more difficult since most of the photos are imperfect, a successfully trained model will however be more robust.

As mentioned earlier, the diversity of a dataset is very important. A dataset should contain a diverse enough set of people of different ethnicities, ages, genders and head poses. Should the dataset for example only contain photos of men, then, in the case of face inpainting, the trained model would not be able to produce correct reconstruction outputs for masked photos of women. This applies for all of the metrics mentioned above. The extreme case described earlier is an example of the so called algorithmic bias, and can occur even in less extreme cases where one of these groups is simply under-represented.

Figure 3.1: Examples of images from the Labelled Faces in the Wild dataset.

Another important feature of a dataset is the general quality of the photos, if all of the photos have good lighting and the subjects are facing the camera directly, the trained model might not be able to correctly reconstruct photos which are not shot under perfect conditions. The key to choosing a good dataset is to pick one which contains some imperfect data, in datasets diversity and balance are very important. Training a model on a balanced dataset will cause the final model to be quite robust and able to generalize well.

3.1.1 Labelled Faces in the Wild

Labelled Faces in the Wild [15] is an in-the-wild dataset which was designed for face recognition. It contains 13233 face images. The images depict 5749 distinct people, 1680 people however have more than two photos in the dataset. The large proportion of people who are present multiple times is well-suited towards face recognition tasks. This could however cause issues in training of a face inpainting network, since the network might skew towards those overrepresented people. Furthermore, the dataset contains a low representation of women and is not very diverse ethnicity-wise. An aligned version of the dataset is provided, the resolution of these aligned images is 250×250 pixels, which by itself would be suitable for use in this thesis. A large share of aligned images are however fairly low-quality and are often cropped in a way which leaves a blank space along the edges of the image. Some examples of images from the dataset are shown in figure 3.1, these images demonstrate the varying quality of the dataset. For the reasons stated above, I decided not to use this dataset.

3.1.2 CelebA

The CelebA dataset [29] is an in-the-wild dataset containing over two hundred thousand photos of celebrities. These photos are of 10177 distinct people. This dataset builds on an earlier dataset called CelebFaces by adding annotations describing individual characteristics of each of the photos. The annotations describe various features of the photo, such as hair colour, whether or not the person is wearing glasses and others. Face landmarks are also provided. These are not of much interest for this thesis, however an aligned and cropped version of this dataset was also made available. The resolution of images in the aligned version of CelebA is 178×218 pixels. This dataset is fairly diverse both in terms of ethnicity and gender, where it however fails to deliver is age. Most of the images are of adults, with children and teenagers being very rare. The elderly are also severely under-represented. The quality of the images is varied, and some low-quality images are present since the aligned dataset was created by extracting faces from larger photos of varying quality, cropping them and then resizing them. The proportion of images where the subject is facing sideways away

Figure 3.2: Examples of images from the CelebA dataset.

Figure 3.3: Examples of images from the CelebA-HQ dataset.

from the camera, or where the face is partially obscured is fairly high. Some of the photos have been cropped and the cropped edges are then filled in with the same pixels as the edge of the crop. This produces odd looking artefacts, as can be seen in figure 3.2. The images are also rectangular, which is somewhat unusual and makes working with the dataset more difficult than with other datasets. I chose not to use this dataset for the reasons mentioned above, but a better version of this dataset is also available, CelebA-HQ.

3.1.3 CelebA-HQ

The CelebA-HQ dataset was created by the authors of Progressive Growing of GANs [22]. The goal behind the creation of this dataset was to provide a high-resolution and high-quality face image dataset for training generative networks. This dataset consists of 30000 high-quality photos of celebrities in a resolution of 1024×1024 pixels. The process of obtaining the high-resolution photographs as described in the paper is as follows. First the image was processed by two pretrained neural networks, the first network was an autoencoder trained for removal of JPEG artefacts and the second one was a GAN based 4x super-resolution network. The resulting image was padded and then a crop area was calculated using the landmarks provided in the original CelebA metadata. This area was resized to 4096×4096 pixels and then downsampled to the final size of 1024×1024 pixels. After this step, 30000 images of the highest quality were selected.

This dataset improves on several issues of the original CelebA dataset, the most important of them is that only high-quality images are present. The images where a significant part of the subject's face is obscured were removed, as were the ones where the subject is not facing the camera. Some images where the subject is facing slightly off-camera are present as shown in figure 3.3. The diversity of this dataset more or less follows the distribution of original CelebA. Gender and ethnicity of subjects are sufficiently balanced while age-wise the distribution is significantly skewed towards adults with children being virtually unrepresented and the elderly being under-represented. I decided to use this dataset for

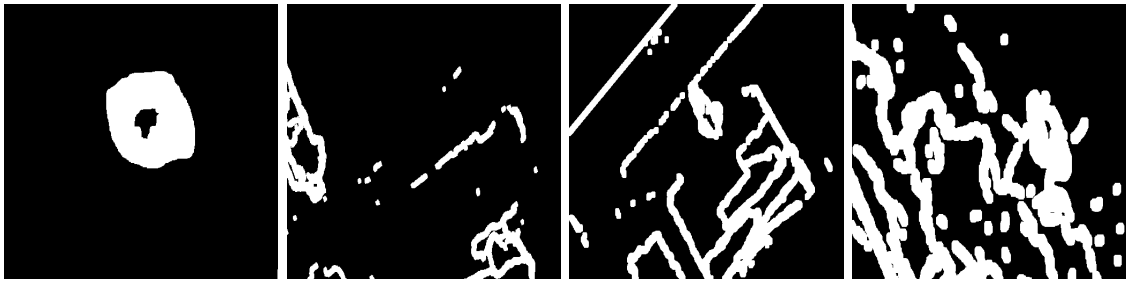

Figure 3.4: Examples of masks from the Irregular masks dataset.

its high quality and because it has been used in several state-of-the-art approaches to face inpainting.

3.1.4 Irregular masks dataset

The irregular masks dataset was specifically developed for the training of free-form inpainting models. It was published along with the paper which introduced partial convolutional layers [28]. This dataset consists of 12000 different irregular binary masks. The masks are provided in a resolution of 512×512 pixels. The dataset contains six categories of masks with different hole-to-image ratios, these are: $(0.01, 0.1]$, $(0.1, 0.2]$, $(0.2, 0.3]$, $(0.3, 0.4]$, $(0.4, 0.5]$, $(0.5, 0.6]$. Each of these categories contains 1000 masks which can reach the border of the image and 1000 masks in which the mask is guaranteed to be at least 50 pixels away from the image border. Some example masks are shown in figure 3.4.

3.2 Tools used

The choice of tools was mostly free by the thesis assignment, except for the choice of programming languages. Python was set as the required language and this makes sense as Python is the current de facto standard for machine learning applications. The next choice to be made was the choice of a machine learning framework. There are several popular frameworks which can be used with Python, the most notable are probably TensorFlow and PyTorch. I chose to use PyTorch because I was already familiar with the way it works and I used it in several university courses previously. While PyTorch is less popular than Tensorflow, it is arguably a simpler framework and it is easier to learn. Furthermore, it seems that PyTorch has been gaining traction in academia and overtook Tensorflow in 2019 in terms of references in academic papers [12].

3.2.1 PyTorch

This section is based on the PyTorch Documentation [41]. PyTorch is a framework which facilitates GPU accelerated machine learning in Python. It is derived from Torch, an older framework which was only available for the Lua programming language. Both Torch and PyTorch were developed by Facebook. As of 2020, only PyTorch is being actively maintained. This section will briefly explain the basics of PyTorch necessary for understanding the implementation of the proposed model. All code is in PyTorch version 1.5.1, the latest version as of writing this thesis.

Tensors Tensors are the basic datatype used in the framework. A tensor is a multidimensional array used for all data-related operations. A tensor can be created either by constructing a new one directly in PyTorch, or by constructing one from a different data type. Methods for creating a tensor from numpy and PIL are available. When tensors are used with neural networks, they are expected to be a four dimensional array. The dimensions then have the following structure:

$$S \times C \times H \times W \tag{3.1}$$

where S is the number of samples, C is the number of channels, H is the height of a sample, W is the width. For example, a minibatch containing four RGB images in a resolution of 512×512 pixels will have the dimensions of $4 \times 3 \times 512 \times 512$. The tensor type supports arithmetic operations and a wide variety of other useful manipulation methods such as concatention and reshaping.

An important feature of PyTorch tensors is automatic differentiation. All operations executed on a tensor are tracked and a complete history of computation is recorded. When computation is finished and gradients are needed for backpropagation, the `.backward()` method can be called to compute all gradients automatically.

Models Every model or a component of a neural network implemented in PyTorch inherits from the `torch.nn.Module` class. To create a model, the user only has to implement the constructor and the `.forward()` method, other methods necessary for the network are inherited. Each model of a network will then be composed of different components, these are defined in the object constructor. The `.forward()` method defines how data flows through the network and is called every time the network is used to evaluate data. PyTorch provides classes for all of the most common basic building blocks of neural networks. As an example, a simple convolutional classifier would be implemented like so:

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, in_features):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 6, 3), # one input channel, 6 output channels, 3x3 convolution
            nn.ReLU(),
            nn.Conv2d(6, 16, 3),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(16 * 6 * 6, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10)
        )
    def forward(x):
        return self.net(x)
```

Listing 3.1: An example of a model implemented in PyTorch.

This network is a classifier intended for the MNIST dataset handwritten number classification, it classifies 32×32 pixel one channel images into one of ten classes. As can be seen, the structure of the network is defined within the constructor of the model object. The basic

components are contained inside the `nn.Sequential` class. This is a container class which simply connects the output of one component to the input of the next. In the example, this class contains two convolutional layers, ReLU activation functions and fully connected layers. The `.forward()` method defines the behaviour of the network when data is to be passed through the network. In the example above, the `.forward()` method simply passes the input of the network to the `nn.Sequential` model and returns its return value.

Training PyTorch slightly differs from other machine learning frameworks in that the users have to write their own training code. There are multiple classes which facilitate neural network training. The first one of these are the optimizers. Optimizers get bound to a neural network model and they can then execute steps of neural network training algorithms, most common optimization algorithms are included in the library, for example Adam and SGD are available. Another vital part in training neural networks are loss functions, just like optimizers the most common ones are provided. Both optimizers and loss functions make use of automatic differentiation to facilitate network training. A single iteration of training a neural network in PyTorch consists of the following steps:

1. Data processing – The network evaluates data inputs and produces outputs
2. Loss computation – The output data is evaluated against the target outputs using a loss function and a loss tensor is returned
3. Gradient computation – The `.backward()` method is called on the loss tensor
4. Optimizer step – The `.step()` method is called on the optimizer object. This causes the optimizer class to execute an iteration of the algorithm it implements.

Hardware acceleration A major feature which makes training of convolutional networks in PyTorch possible is the use of hardware acceleration. PyTorch is able to use the CUDA and cuDNN parallel libraries by NVidia. CUDA is a library which allows the user to run arbitrary mathematical code on an Nvidia GPU. The cuDNN library builds on the CUDA API and offers GPU-accelerated primitives specifically for the training of deep neural networks. PyTorch uses these libraries to significantly improve the performance of training by using the parallel computing power of GPUs. Tensors and models can be moved to the memory of the GPU using the `.to()` method.

3.2.2 Computing resources

Since training of neural networks is very computationally intensive, GPUs are nowadays the only viable way of achieving reasonable performance. In the initial phase of development of this thesis I used my own computer's GPU for training. Since it is an Nvidia GTX 1060, an older mid-range GPU, training of more complex models was quickly becoming unmanageable. Apart from my own computer I used two different services for training, namely Google Colab and MetaCentrum.

Google Colaboratory The first service I used was Google Colaboratory[1]. This is a cloud-based computing service which allows users to run GPU accelerated code on servers provided by Google for free. There are however several disadvantages to using Colaboratory,

[1] https://colab.research.google.com/

the first one of which is that all programs ran on this service have to be in the Jupyter Notebook format, which can pose some issues regarding code readability and manageability for larger programs. Furthermore all data accessed by the program must be uploaded to Google Drive, which due to its limited size further restricts the usability of Colaboratory. The last and the main issue is that each session on Colaboratory is limited to runtime of about 12 hours and the browser window in which the session is running has to be kept open. Even though Colaboratory offers impressive hardware, the time limitation basically prevents the user from training a more complex model. There is a premium version of Colaboratory which removes the time and activity session constraints, but it is unfortunately not available for users in Europe. For these reasons using Colaboratory proved too difficult to be useful for my purposes and a different service had to be used.

MetaCentrum Fortunately my thesis supervisor informed me of the existence of MetaCentrum[2]. MetaCentrum is a department of CESNET, the developer of regional network infrastructure in the Czech Republic, it coordinates computational resources provided by universities and research institutes into a grid computing service. Access is provided free of charge to students and academic workers. The range of servers on which GPU accelerated computation can be done includes three servers equipped with Nvidia Tesla GPUs and one with Nvidia Titan. These are top of the line GPUs designed specifically for hardware accelerated computing.

The grid uses a scheduling system to assign computing power on demand. When a user wants to run longer-running code a batch file performing all necessary operations has to be created and submitted to a queue. From the time the request was submitted, this batch file along with information provided at submission is referred to as a job. The scheduling system then runs this job whenever computing resources are free and no jobs with a higher priority are in the same queue. There are multiple queues grouped by the properties of the code to be run, the ones of most interest to me were the `gpu` queue which is used for jobs which require GPU acceleration and run for less than 24 hours and the `gpu_long` queue for GPU accelerated jobs with a runtime of up to one week. Waiting times from submitting a job to it actually being ran range from minutes to hours for the `gpu` queue and hours to days for the `gpu_long` queue.

During development I ran into several issues with the compatibility of PyTorch and the different servers on which my code was being ran. Different versions of PyTorch require different specific versions of CUDA and cuDNN libraries. Even though the MetaCentrum system allows the user to import modules containing specific versions of these libraries, I was never able to actually find a correct working configuration. After contacting the staff of MetaCentrum, I was able to solve these compatibility issues by using a container image. These images are provided by Nvidia at their NVIDIA GPU Cloud website[3]. Containers for PyTorch are provided with all necessary frameworks and libraries. Initially I used image version 19.02, since it was the last one which used CUDA version 10.0 and choosing an image running a higher version would exclude one of MetaCentrum servers because it used outdated GPU drivers. Later I had to move to version 20.03, because of issues regarding an older version of PyTorch and implementing spectrally normalized convolutions. For details on these realeases, please see the release notes of PyTorch containers[4]. These images were

[2] https://metavo.metacentrum.cz/

[3] https://ngc.nvidia.com/

[4] https://docs.nvidia.com/deeplearning/frameworks/pytorch-release-notes/index.html

then ran using Singularity[5]. Singularity is a container platform which allows the user to run programs provided in the image while still allowing access to the local filesystem outside the image. After transferring to the use of Singularity I did not encounter any more compatibility issues.

3.3 Proposed solution

I chose to base my solution on a convolutional GAN network with a generator based on an U-Net like architecture. This choice was mostly driven by currently state-of-the-art solutions to face inpainting, like for example the SC-FEGAN[19] which showed that a generator employing the U-Net structure in combination with gated convolutions is very effective. Many different models were tried and modified until the model which will be presented in the next sections was developed.

3.3.1 Generator

As discussed earlier, autoencoders and U-Nets are the most common network structures used for face inpainting. I chose to go with an U-Net whose structure will be presented after an overview of the implementation details of its vital components

Gated convolutional layer The implementation of gated convolutional layers was fairly straightforward as the authors of the original paper have provided both an in-depth description of the layer in the paper and a GitHub repository[6] with an implementation in Tensorflow. The implemented layer is as follows:

```
class GatedConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0,
                 dilation=1, groups=1, bias=True, padding_mode='zeros'):
        super(GatedConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=stride,
                              padding=padding, dilation=dilation, groups=groups, bias=bias,
                              padding_mode=padding_mode)
        self.mask_conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=stride,
                              padding=padding, dilation=dilation, groups=groups, bias=bias,
                              padding_mode=padding_mode)
        self.sigmoid = nn.Sigmoid()
        self.relu = nn.ReLU()
        self.bn = nn.BatchNorm2d(out_channels)

    def forward(self, in_x):
        x = self.relu(self.conv(in_x))
        mask = self.sigmoid(self.mask_conv(in_x))
        return self.bn(x * mask)
```

Listing 3.2: The implementation of the gated convolutional layer.

As can be seen, the layer contains two convolutional layers, `conv` is used for feature extraction and `mask_conv` is used to extract the learned mask. The `forward` method computes the output of the layer as per the formula shown in 2.4.3. Because every gated convolutional layer in the implemented model is followed by batch normalization, a batch normalization layer was added into this object for simplicity. Interestingly, the original

[5] https://sylabs.io/docs/
[6] https://github.com/JiahuiYu/generative_inpainting

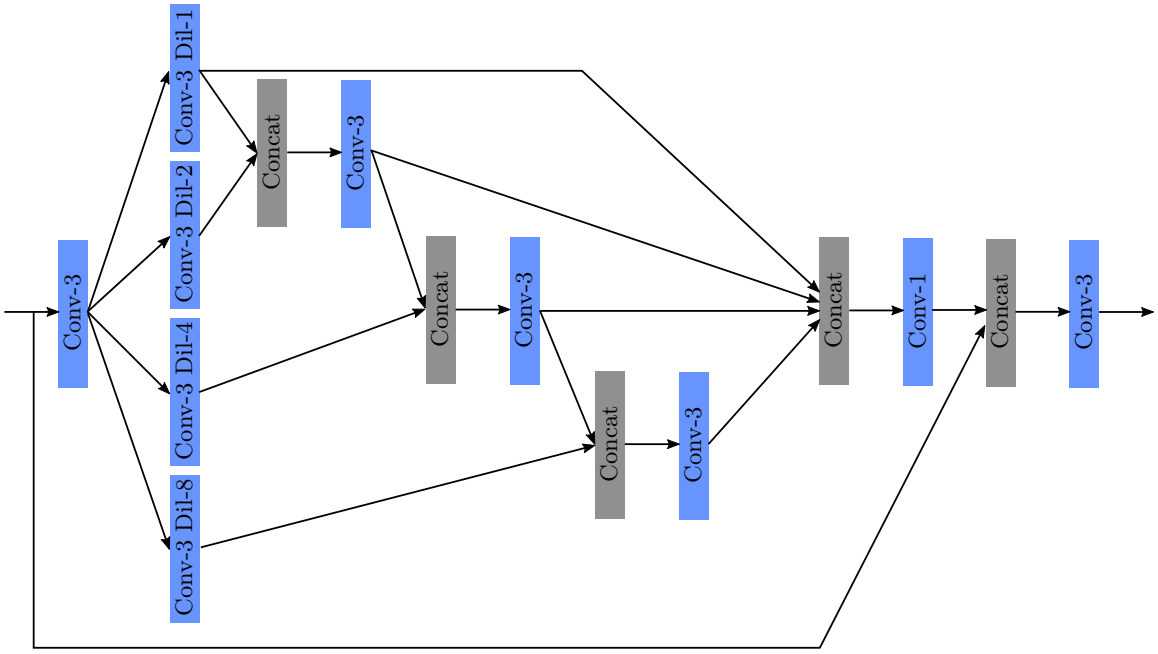

Figure 3.5: A diagram of the modified dense multiscale fusion block. "Conv-3 Dil-2" represents a convolutional layer with a kernel size of 3×3 and dilation of 2.

paper implements transposed gated convolution as $2\times$ upscaling followed by a regular convolutional layer. This is commonly done to avoid square artefacts which are produced by transposed convolution.

Dense multiscale fusion blocks The version of dense multiscale fusion blocks used in the proposed model is modified by replacing conventional convolutional layers by gated ones. Code for this block will not be presented here because it is fairly long and is better represented by the previously shown figure 2.8. All layers are followed by ReLU activation and batch normalization.

Another modified version was also created, shown in figure 3.5. In this version, element-wise additions were replaced by concatenations and one more gated convolutional layer was added at the end of the block in order to merge the concatenation of the original input data and data processed by the inner part of the block. Because concatenating two tensors results in a tensor with double the number of channels, the inner convolutions which directly follow concatenations are reducing the number of channels back to the original number in order to keep the number of channels inside the block low. Even though dense multiscale fusion blocks were introduced after gated convolutions, they have never been used together. The modified DMFB architecture has been designed specifically for use with gated convolutional layers. The motivation behind the use of concatenation instead of element-wise addition is to try to better utilize the learned masks used in gated convolutions. A gated convolution could learn to merge features more efficiently than pure addition. The performance of these blocks will be evaluated in the next chapter.

Network structure The structure of the generator network is shown in table 3.1. It is an U-Net which uses a two-step encoder-decoder structure. The input image is downsampled twice using convolutions with a stride of 2. The inner part of the network then works with

Layer	Kernel	Output shape	Comment
Input		$(4, H, W)$	Concatenated masked image and mask
GatedConv	5	$(64, H, W)$	Skip connection 1
GatedConv	3	$(128, H/2, W/2)$	
GatedConv	3	$(128, H/2, W/2)$	Skip connection 2
GatedConv	3	$(256, H/4, W/4)$	
GatedConv	3	$(256, H/4, W/4)$	Skip connection 3
GatedConv	3	$(256, H/4, W/4)$	
DMFB		$(256, H/4, W/4)$	64 inner channels
DMFB		$(256, H/4, W/4)$	64 inner channels
DMFB		$(256, H/4, W/4)$	64 inner channels
DMFB		$(256, H/4, W/4)$	64 inner channels
GatedConv	3	$(256, H/4, W/4)$	
Concat		$(512, H/4, W/4)$	Skip connection 3
GatedConv	3	$(256, H/4, W/4)$	
GatedConv	3	$(256, H/4, W/4)$	
GatedDeconv	3	$(128, H/2, W/2)$	
GatedConv	3	$(128, H/2, W/2)$	
Concat		$(256, H/2, W/2)$	Skip connection 2
GatedConv	3	$(128, H/2, W/2)$	
GatedConv	3	$(128, H/2, W/2)$	
GatedDeconv	3	$(64, H, W)$	
GatedConv	3	$(64, H, W)$	
Concat		$(128, H, W)$	Skip connection 1
GatedConv	3	$(64, H, W$	
GatedConv	3	$(64, H, W)$	
GatedConv	3	$(32, H, W)$	
Conv	3	$(3, H, W)$	Conventional convolution + Tanh

Table 3.1: The structure of the proposed generator.

data which is a quarter of the original size in each dimension. The input number of channels is 4, three of those are the RGB masked input image and one is the mask. The number of channels is then gradually increased until it reaches 256. This number of channels is then used in the inner layer where four dense multiscale fusion blocks are used. The DMFBs then use 64 channels in their inner convolutions. After the data has been processed in the inner part, the dimensions are gradually increased back to their original size while the number of channels is gradually decreased. All layers except the last are gated convolutions with ReLU activations and batch normalization, the final layer of this model is a vanilla convolution rather than a gated convolution. This layer is followed by hyperbolic tangent activation and produces the final 3-channel output, no normalization is used. Three skip connections are used to propagate data through the network.

3.3.2 Discriminator

I chose to use a SN-PatchGAN based discriminator. The initial structure was based on the discriminator used in the pix2pix network[18] which was the first one to introduce the concept of patch-based discriminators as discussed earlier. My implementation used vanilla

Layer	Kernel	Output shape	Comment
Input		$(4, H, W)$	Concatenated masked image and mask
SNConv	4	$(64, H/2, W/2)$	LeakyReLU(0.2) activation
SNConv	4	$(128, H/4, W/4)$	LeakyReLU(0.2) activation
SNConv	4	$(256, H/16, W/16)$	LeakyReLU(0.2) activation
SNConv	4	$(256, H/32, W/32)$	LeakyReLU(0.2) activation
SNConv	4	$(256, H/32, W/32)$	Sigmoid activation

Table 3.2: The structure of the proposed discriminator.

convolutions in combination with batch normalization. This version worked rather poorly and mode collapsed several times. The issues with mode collapse were solved by the use of spectral convolutions.

Spectral normalized convolution As can be seen in listing 3.3, the spectral normalized convolution is composed of a single vanilla convolution. In the constructor, the convolutional layer object is passed as an argument to the `spectral_norm()` function. This function applies what is referred to as a "hook" to the layer object. A hook applies a function to the layer object and this function is called every time the `.forward()` method is called. The call of the hooked function is executed before the code of the `.forward()` method. In this case the weights of the layer are rescaled according to the description provided in 2.4.5.

```
class SpectralNormConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0,
                 dilation=1, groups=1, bias=True, padding_mode='zeros'):
        super(SpectralNormConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=stride,
                    padding=padding, dilation=dilation, groups=groups, bias=bias,
                    padding_mode=padding_mode)
        self.conv = torch.nn.utils.spectral_norm(self.conv)

    def forward(self, x):
        return self.conv(x)
```

Listing 3.3: The implementation of the spectral normalized convolutional layer.

Network structure The structure of the discriminator network is shown in table 3.2. The discriminator consists of five spectral normalized convolutional layers. The initial four layers have a kernel size of 4 and stride of 2. Each of these layers halves the width and height of the processed data. These layers are followed by LeakyReLU activation with a slope of 0.2. The last layer also has a kernel size of 4, but it keeps all dimensions the same. It is followed by Sigmoid activation in order to squash the values of the output data into an interval between zero and one. The four channel input of the network is composed of three channels of RGB data either from the dataset or produced by the generator concatenated with a one channel mask. The classification output is a 256 channel three dimensional array, meaning that there are 256 classifications for each patch within the image. Each value of the output is a prediction of whether the data seems real or fake.

3.3.3 Training

The training data is a combination of a sample from the training and a binary mask. No preprocessing is done on the image dataset. The mask dataset is preprocessed to a PNG where the mask is represented by a maximum value in all channels including the alpha channel, no mask is represented by zero in all channels. The inputs are normalized from the range of $[0, 1]$ to $[-1, 1]$ during training. The data is shuffled in every epoch in an effort to combat overfitting.

The Adam optimizer was used with a learning rate of 0.0002. The values of β_1 and β_2 were set to $\beta_1 = 0.5$ and $\beta_2 = 0.999$. These values are used in the calculation of the decay rates of the first and second moment used in the algorithm. This setting was found to perform well, other values which were tested were $\beta_1 = 0.9$, $\beta_2 = 0.99$ and their combinations. Other values of learning rates were not tested since the value which was used is commonly used in training of other inpainting models. The Adam optimizer with these settings was used for training of both the generator and the discriminator.

For the training of the generator network a combined loss was used. This loss consists of three loss functions which where used with equal weighting. The first loss is a conventional adversarial GAN loss, the only difference being the target values. Since the discriminator is a PatchGAN based one, the target values of the discriminator output are an array of ones rather than a singular value. Binary cross-entropy was used as the loss function for adversarial loss. The other two losses both use L1 loss. The first one of those is computed across the whole image, giving the global loss term. The second one is only computed from the regions covered by the mask, giving the local loss term. The intuition behind using the local loss is that the trained network will be penalized more significantly for incorrect inpainting results. The combined loss can be described as:

$$L_{total} = \frac{L_{global} + L_{local} + L_{adv}}{3} \tag{3.2}$$

where L_{global} is the global L1 loss, L_{local} is the local L1 loss and L_{adv} is the adversarial loss.

The discriminator network was trained using the binary cross-entropy loss. The total loss term consists of two loss values. One is computed for the inputs which were generated by the generator network, the "fake" images. The target network output for these inputs is an array of zeros. The "real" images taken from the training dataset have a target of an array of ones.

Chapter 4

Solution evaluation

In this chapter, three of the implemented models will be presented and compared, later an evaluation of their performance will be given. These are the three models which performed the best of the ones which were tested. All three models follow the same U-Net like structure and use the SN-PatchGAN discriminator architecture as presented in 3.3. The models are the following:

Model 1 The first model can be seen as a baseline of sorts for the purposes of evaluating the results. This model is of the U-Net structure and uses dense multiscale fusion blocks in the original structure as described in section 2.4.4. The only modification is that vanilla convolutions are replaced by gated convolutions. The model is trained using the combined loss as shown in formula 3.2.

Model 2 The second model uses the same structure as the first model, except for the DMFBs. The modified version of the dense multiscale fusion blocks is used. The structure of the modified block can be seen in 3.5. The motivation behind modifying the blocks was to better utilize the fact that gated convolutions learn a separate mask for each channel. My theory is that the use of element-wise addition for intermediate results within the original block resulted in some information loss. In the modified version concatenations are used instead, thus the inner gated convolutions which then merge the intermediate information could learn masks in such a way that could make the merging of the extracted features more effective. The same combined loss was used as in the first model.

Model 3 The third model is exactly the same as Model 2 in terms of architecture. The difference is that perceptual loss was used in addition to the loss functions which were used by the previous models. The combined loss with the addition of the loss term is the following:

$$L_{total} = \frac{L_{global} + L_{local} + L_{adv} + 0.05L_{percept}}{3.05} \tag{4.1}$$

where $L_{percept}$ is the perceptual loss. The fairly low weight of the perceptual loss term is inspired by SC-FEGAN.

Model	Training time
Model 1	36:28:35
Model 2	41:30:59
Model 3	65:33:02

Table 4.1: Training times of the evaluated models. Time is in hh:mm:ss format.

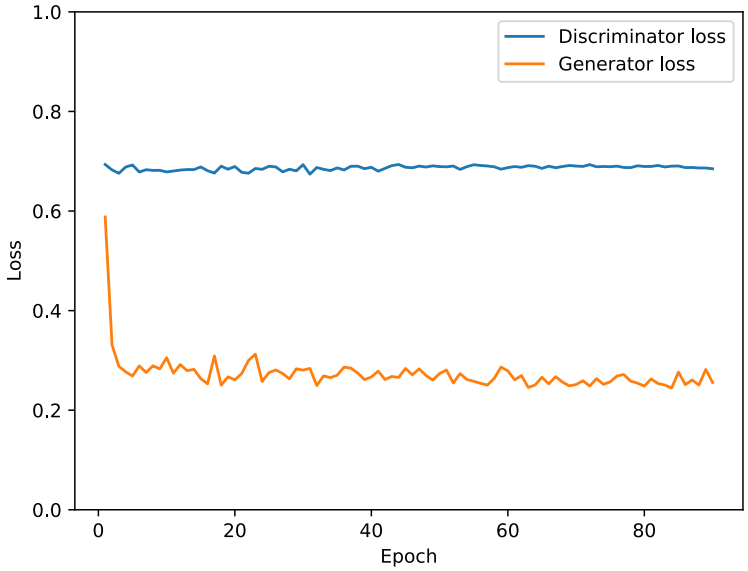

Figure 4.1: A graph showing training losses of Model 2 during training.

4.1 Training

The training datasets which were used were the CelebA-HQ dataset and the Irregular masks dataset. The images were rescaled to 128×128 pixels. A 90/10 training-testing split was used for the face image data, giving 27000 training samples and 3000 testing samples. As for the mask dataset, I did not feel that there was a danger of the generator network remembering the masks and hard-learning the data which should be filled in. Because of this, a testing dataset of masks was instead created by copying 500 masks of each of the hole-to-image ratio categories. The fairly low number of samples in the mask dataset led to the use of data augmentation in the form of flipping the mask horizontally with a 50% probability.

The proposed models were trained for 90 epochs with a batch size of 4. The fairly small batch size is inspired by face inpainting papers which commonly use small batches, for example the original partial convolution paper uses a batch size of 6 [28]. All of the models were ran on the same MetaCentrum server which uses the Nvidia Tesla T4 GPU. The training times of the models are shown in table 4.1. A graph of the training losses during training is shown in figure 4.1. Note that the generator loss shown in the graph is the combined loss. A fairly interesting property of spectral normalized discriminators can be seen in the graph as the training loss stays nearly constant during training.

Mask coverage	Model 1	Model 2	Model 3
$(0.01, 0.1]$	0.051	0.031	0.049
$(0.1, 0.2]$	0.049	0.036	0.050
$(0.2, 0.3]$	0.062	0.045	0.057
$(0.3, 0.4]$	0.069	0.056	0.065
$(0.4, 0.5]$	0.080	0.069	0.081
$(0.5, 0.6]$	0.112	0.102	0.122
Average	0.070	**0.056**	0.071

Table 4.2: A comparison of the proposed models by L1 loss.

Mask coverage	Model 1	Model 2	Model 3
$(0.01, 0.1]$	20.12	26.04	19.75
$(0.1, 0.2]$	20.64	24.01	19.64
$(0.2, 0.3]$	18.38	21.28	19.03
$(0.3, 0.4]$	17.21	19.18	17.73
$(0.4, 0.5]$	16.22	17.41	16.09
$(0.5, 0.6]$	13.64	14.16	12.99
Average	17.70	**20.35**	17.54

Table 4.3: A comparison of the proposed models by PSNR.

4.2 Evaluation metrics

I chose to use L1 loss and PSNR for evaluating the performance of the models. Peak signal to noise ratio (PSNR) is used as a relative measure of quality of reconstruction of an image when compared to the original. It is commonly used for comparing compression algorithms, but in the case of face inpainting it is a fitting measure as well. The formula for computing PSNR is as follows:

$$PSNR = 10 \log_{10} \left(\frac{1}{MSE} \right) \tag{4.2}$$

where MSE is the mean square error between the reconstructed image and the original image. L1 loss has already been presented in section 2.2.2. For L1 loss a lower value means a better result, while for PSNR a higher value means a better result.

4.3 Results

The testing subset of the mask dataset was grouped by the hole-to-image ratio for the purposes of evaluating the models. Tables 4.2 and 4.3 show performance of the models measured by L1 loss and PSNR respectively. As can be seen, Model 2 which used the modified dense multiscale fusion blocks performed the best in both metrics. Rather surprisingly, Model 3 performed the worst on PSNR and was on the same level as Model 1 in terms of L1 loss. The addition of perceptual loss worsened the overall performance when compared to Model 2 and almost doubled the training time.

A comparison of the tested models is shown in figure 4.2. Model 1 seems to work well, but it sometimes produces distortions in terms of edges of faces, see first and second image. Model 3 is fairly accurate edge-wise, it however produces odd-looking pinkish skin color. Lips of people who are smiling in the photos are also distorted.

Figure 4.2: A comparison of images produced by the tested models.

It is visible that Model 2 performs the best of the three. It produces no discolouration in terms of skin tone and it produces high-frequency details such as hair fairly well. The model also works well in terms of generation of a new face when almost all of the input is obscured by a mask, for an example see image 4. In this image almost the entire area of the face is obscured by the mask, the only information that is given to the network is the hair colour, the shape of the chin and a part of the nose. The model is able to produce a high quality reconstruction even with this little information. The performance advantage of Model 2 is also clearly visible in image row 5. In this case the model was able to reconstruct the fringe hair with very little distortion when compared to the other two models.

Chapter 5

Conclusions

The goal of this thesis was to design a neural network for face image reconstruction. A modified dense multiscale fusion block architecture was designed specifically for the use with gated convolutional layers. This modified block architecture was then used as a part of an U-Net based generator network in combination with a spectrally normalized patch-based discriminator. As shown by the performed experiments, networks which use the modified block architecture outperformed the networks which used blocks with the original architecture. The model developed in this thesis shows that dense multiscale fusion blocks can be modified so that their performance is increased when used in a network which uses gated convolutional layers. The current state-of-the art approaches based on the use of gated convolutional layers generally use a plain U-Net based or an autoencoder based architecture. This thesis has successfully shown that these two approaches can be combined and the resulting network can be used for face inpainting tasks.

The resulting model designed in this thesis was able to successfully produce inpainting outputs on data from the testing set in a resolution of 128×128 pixels. The network which uses the modified architecture of multiscale blocks is able to generate images with an average PSNR of 20.32 while the baseline network reaches an average PSNR of 17.70. The difference in inpainting quality is even more significant at low mask-to-image ratios where the baseline network reaches PSNR of 20.12 and the network which uses the modified architecture generates images with PSNR of 26.04. The network using the modified architecture also consistently outperforms the baseline in average L1 loss. Additional network output samples are provided in appendix B.

Further increasing the depth of the network and the number of blocks used in the model could allow the network to be used with data in greater resolution, however training a deeper network would result in extending the already long training times. Further improving the quality of generated images could be possible with the use of additional loss functions in the combined loss, however a combination which would perform better than the one used for the model presented in this thesis was not found.

Bibliography

[1] Adiga, S., Attia, M. A., Chang, W. and Tandon, R. ON THE TRADEOFF BETWEEN MODE COLLAPSE AND SAMPLE QUALITY IN GENERATIVE ADVERSARIAL NETWORKS. In: *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. Nov 2018, p. 1184–1188. ISSN null.

[2] Anwarul, S. and Dahiya, S. A Comprehensive Review on Face Recognition Methods and Factors Affecting Facial Recognition Accuracy. In: Singh, P. K., Kar, A. K., Singh, Y., Kolekar, M. H. and Tanwar, S., ed. *Proceedings of ICRIC 2019*. Cham: Springer International Publishing, 2020, p. 495–514. ISBN 978-3-030-29407-6.

[3] AT&T Laboratories Cambridge. *Eigenfaces* [online]. Wikipedia, march 2004 [cit. 2020-01-15]. Available at: https://en.wikipedia.org/wiki/File:Eigenfaces.png.

[4] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

[5] Brownlee, J. *A Gentle Introduction to Generative Adversarial Networks (GANs)* [online]. Machine Learning Mastery, june 2019 [cit. 2020-01-16]. Available at: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/.

[6] Cosgrove, C. *Spectral Normalization Explained* [online]. Ode to Gradients, january 2018 [cit. 2020-07-13]. Available at: https://christiancosgrove.com/blog/2018/01/04/spectral-normalization-explained.html.

[7] Dalal, N. and Triggs, B. Histograms of oriented gradients for human detection. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. June 2005, p. 886–893 vol. 1. ISSN 1063-6919.

[8] Dumoulin, V. and Visin, F. *A guide to convolution arithmetic for deep learning*. 2016.

[9] Glorot, X., Bordes, A. and Bengio, Y. Deep Sparse Rectifier Neural Networks. In: Gordon, G., Dunson, D. and Dudík, M., ed. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, p. 315–323. Proceedings of Machine Learning Research, vol. 15. Available at: http://proceedings.mlr.press/v15/glorot11a.html.

[10] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. Available at: http://www.deeplearningbook.org.

[11] Goodfellow, I. J., Pouget Abadie, J., Mirza, M., Xu, B., Warde Farley, D. et al. *Generative Adversarial Networks*. 2014.

[12] He, H. *The State of Machine Learning Frameworks in 2019* [online]. The Gradient, october 2019 [cit. 2020-7-16]. Available at: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

[13] He, K., Zhang, X., Ren, S. and Sun, J. *Deep Residual Learning for Image Recognition.* 2015.

[14] He, K., Zhang, X., Ren, S. and Sun, J. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* 2015.

[15] Huang, G. B., Ramesh, M., Berg, T. and Learned Miller, E. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments.* 07-49. University of Massachusetts, Amherst, October 2007.

[16] Hui, Z., Li, J., Wang, X. and Gao, X. *Image Fine-grained Inpainting.* 2020.

[17] Ioffe, S. and Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015.

[18] Isola, P., Zhu, J.-Y., Zhou, T. and Efros, A. A. *Image-to-Image Translation with Conditional Adversarial Networks.* 2016.

[19] Jo, Y. and Park, J. *SC-FEGAN: Face Editing Generative Adversarial Network with User's Sketch and Color.* 2019.

[20] Johnson, J., Alahi, A. and Fei Fei, L. *Perceptual Losses for Real-Time Style Transfer and Super-Resolution.* 2016.

[21] Kanade, T. Computer Recognition of Human Faces. *Interdisciplinary Systems Research.* January 1977, vol. 47.

[22] Karras, T., Aila, T., Laine, S. and Lehtinen, J. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* 2017.

[23] Kingma, D. P. and Ba, J. *Adam: A Method for Stochastic Optimization.* 2014.

[24] Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S. *Self-Normalizing Neural Networks.* 2017.

[25] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105. NIPS'12.

[26] Lecun, Y. Generalization and network design strategies. In: Pfeifer, R., Schreter, Z., Fogelman, F. and Steels, L., ed. *Connectionism in perspective.* Elsevier, 1989.

[27] Lecun, Y., Bottou, L., Orr, G. and Müller, K.-R. Efficient BackProp. august 2000.

[28] Liu, G., Reda, F. A., Shih, K. J., Wang, T.-C., Tao, A. et al. *Image Inpainting for Irregular Holes Using Partial Convolutions.* 2018.

[29] Liu, Z., Luo, P., Wang, X. and Tang, X. Deep Learning Face Attributes in the Wild. In: *Proceedings of International Conference on Computer Vision (ICCV)*. December 2015.

[30] Lu, L., Shin, Y., Su, Y. and Karniadakis, G. E. *Dying ReLU and Initialization: Theory and Numerical Examples*. 2019.

[31] Maas, A. L., Hannun, A. Y. and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In: *Proc. icml*. 2013, p. 3.

[32] Prechelt, L. Early Stopping - But When? march 2000.

[33] Radford, A., Metz, L. and Chintala, S. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015.

[34] Ramachandran, P., Zoph, B. and Le, Q. V. Searching for Activation Functions. *CoRR*. 2017, abs/1710.05941. Available at: http://arxiv.org/abs/1710.05941.

[35] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. *You Only Look Once: Unified, Real-Time Object Detection*. 2015.

[36] Ronneberger, O., Fischer, P. and Brox, T. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015.

[37] Simonyan, K. and Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014.

[38] Sirovich, L. and Kirby, M. Low-dimensional procedure for the characterization of human faces. *J. Opt. Soc. Am. A*. OSA. Mar 1987, vol. 4, no. 3, p. 519–524. Available at: http://josaa.osa.org/abstract.cfm?URI=josaa-4-3-519.

[39] Szeliski, R. *Computer Vision: Algorithms and Applications*. 1stth ed. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 1848829345.

[40] Taigman, Y., Yang, M., Ranzato, M. and Wolf, L. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, p. 1701–1708.

[41] Torch contributors. *PyTorch documentation* [online]. PyTorch, june 2019 [cit. 2020-06-16]. Available at: https://https://pytorch.org/docs/1.5.0/.

[42] Turk, M. and Pentland, A. Eigenfaces for Recognition. *J. Cognitive Neuroscience*. Cambridge, MA, USA: MIT Press. january 1991, vol. 3, no. 1, p. 71–86. Available at: https://doi.org/10.1162/jocn.1991.3.1.71. ISSN 0898-929X.

[43] Viola, P. and Jones, M. Rapid Object Detection using a Boosted Cascade of Simple Features. In:. February 2001, p. I–511. ISBN 0-7695-1272-0.

[44] Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y. et al. *ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks*. 2018.

[45] Xu, B., Wang, N., Chen, T. and Li, M. *Empirical Evaluation of Rectified Activations in Convolutional Network*. 2015.

[46] You, K., Long, M., Wang, J. and Jordan, M. I. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019.

[47] Yu, F. and Koltun, V. *Multi-Scale Context Aggregation by Dilated Convolutions.* 2015.

[48] Yu, J., Lin, Z., Yang, J., Shen, X., Lu, X. et al. *Free-Form Image Inpainting with Gated Convolution.* 2018.

Appendix A

SD card contents

- `/thesis` – Contains this thesis in pdf format.
- `/thesis/src` – Contains the Latex source of this thesis.
- `/src` – Contains the Python source code.
- `/models` – Contains trained models which were presented in chapter 4.
- `/datasets/celebAHQ` – Contains the CelebA-HQ dataset scaled to 128×128 pixels.
- `/datasets/masks` – Contains the Irregular masks dataset scaled to 128×128 pixels.
- `/containers` – Contains the NGC PyTorch singularity container.
- `/samples` – Contains images generated from the testing dataset by each of the three presented models.
- `/README.md` – Contains additional description of the provided files and instructions on how to run the provided programs.

Appendix B

Additional output samples

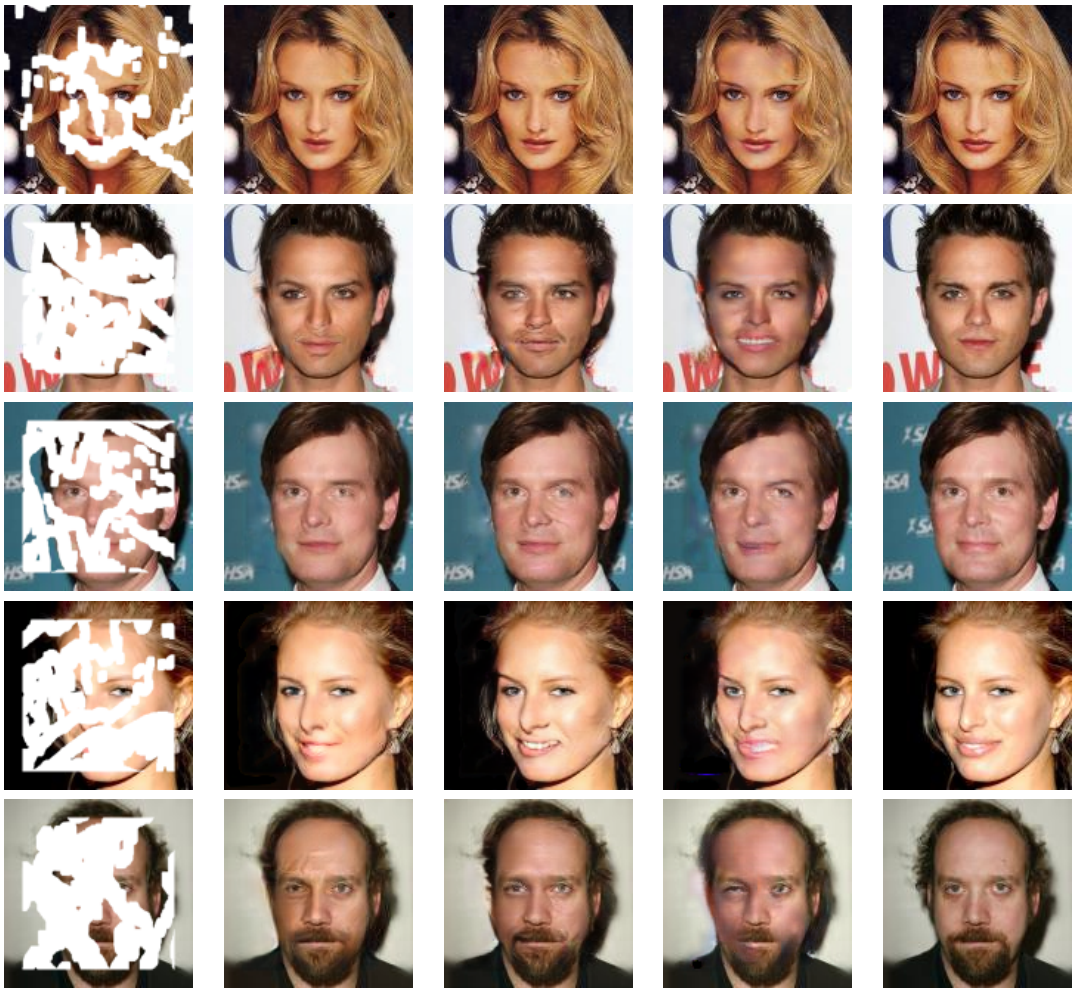

Figure B.1: A comparison of images produced by the tested models.

Figure B.2: Examples of outputs of the network which uses modified dense multiscale fusion blocks.