Thomas Neff, BSc.

Data Augmentation in Deep Learning using Generative Adversarial Networks

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. DI Dr. Horst Bischof
Institute of Computer Graphics and Vision, Graz University of Technology

Advisor

DI Dr. Martin Urschler
Ludwig Boltzmann Institute for Clinical Forensic Imaging, Graz

Graz, Austria, March 2018

Abstract

In recent years, deep learning based methods achieved state-of-the-art performance in most computer vision tasks. However, these methods are typically supervised, and require huge amounts of annotated data to train. Acquisition of annotated data can be a costly endeavour, especially for pixelwise methods such as image segmentation. To circumvent these costs and train on smaller datasets, data augmentation is commonly used to artificially generate additional training data. A major downside of standard data augmentation methods is that they require knowledge of the underlying task in order to perform well, and introduce additional hyperparameters into the deep learning setup.

To improve on these issues, we propose a novel method of data augmentation utilizing Generative Adversarial Networks (GANs). By modifying the *GAN*-formulation to generate image-segmentation pairs, we can train a generative model that synthesizes new images and their corresponding segmentation masks from random noise. These synthetic image-segmentation pairs can then further be used to train segmentation networks, effectively acting as a data augmentation method.

We evaluate our method on two image segmentation tasks: medical image segmentation of the left lung of the SCR Lung Database and semantic segmentation of the Cityscapes dataset. For the medical segmentation task, we show that our *GAN*-based augmentation performs as well as standard data augmentation, and training on purely synthetic data even outperforms our previously published results. For the Cityscapes evaluation, we report that our *GAN*-based augmentation scheme is competitive with standard data augmentation methods, only performing slightly worse. We show synthetic image-segmentation pairs for both datasets and demonstrate that even for complex datasets such as Cityscapes, our *GAN* manages to generate reasonable synthetic data, suggesting that *GAN*-based augmentation has potential for future research.

Keywords. Generative Adversarial Networks, data augmentation, segmentation, deep learning, medical image analysis

Kurzfassung

Methoden basierend auf Deep-Learning haben in den letzten Jahren in den meisten Anwendungen im Bereich Computer-Vision den Stand der Technik übertroffen. Viele dieser Methoden beruhen auf überwachtem Lernen, und benötigen daher riesige Mengen an annotierten Daten um gute Ergebnisse zu erzielen. Die Beschaffung von annotierten Daten kann ein kostspieliges Unterfangen sein, vor allem für pixelweise Methoden, wie zum Beispiel Bildsegmentierung. Um diese Kosten zu umgehen, und auch mit einer geringen Anzahl an Daten trainieren zu können, wird häufig Datenaugmentierung verwendet, um künstlich neue Daten zu erzeugen. Ein großer Nachteil von konventioneller Datenaugmentierung ist, dass diese zusätzliches Wissen über die zugrunde liegenden Daten und deren Anwendung voraussetzt, um gute Ergebnisse zu erzielen, sowie dass durch Datenaugmentierung zusätzliche Hyperparameter in das Deep-Learning-Setup eingebracht werden.

Wir präsentieren eine neuartige Methode der Datenaugmentierung, basierend auf Generative Adversarial Networks (GANs), deren Ziel es ist, typische Nachteile der Datenaugmentierung zu beheben. Unsere Methode modifiziert die Struktur der *GANs*, sodass Paare bestehend aus Bild und dazugehöriger Segmentierungsmaske aus Rauschen erzeugt werden. Diese Paare können weiterführend genutzt werden, um Segmentierungsnetzwerke zu trainieren, wodurch sie effektiv als Datenaugmentierungsmethode fungieren.

Wir evaluieren unsere Methode auf zwei Anwendungen im Bereich der Bildsegmentierung: medizinische Bildsegmentierung des linken Lungenflügels basierend auf den Daten der ‘SCR Lung Database’, sowie semantische Bildsegmentierung basierend auf dem Datensatz ‘Cityscapes’. Wir zeigen, dass unsere Methode der *GAN*-basierten Datenaugmentierung für die medizinische Bildsegmentierung gleichwertige Ergebnisse erzielt, wie konventionelle Datenaugmentierung, und dass Netzwerke, die rein auf synthetischen Daten trainiert wurden, bessere Ergebnisse erzielen als in unseren bisher publizierten Resultaten. In der Evaluierung des Cityscapes Datensatzes zeigen wir, dass unsere *GAN*-basierte Datenaugmentierunsmethode konkurrenzfähig mit konventioneller Datenaugmentierung ist, und nur leicht schlechtere Ergebnisse erzielt. Zudem zeigen wir synthetische Bilder und dazugehörige Segmentierungsmasken für beide Anwendungen, und demonstrieren, dass unser *GAN* sogar für komplexe Datensätze wie Cityscapes plausible synthetische Daten erzeugen kann. Dieses Ergebnis legt nahe, dass *GAN*-basierte Datenaugmentierung Potenzial für zukünftige Forschung zeigt.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Ort	Datum	Unterschrift

Acknowledgments

First of all, I'd like to thank my advisor, DI Dr. Martin Urschler. Throughout the last couple of years, you have been an amazing mentor to me, and managed to push me towards my strengths while still allowing me to independently work on my thesis projects. Thank you for introducing me to the exciting topic of deep learning in the domain of medical imaging. It was great to present our work at the OAGM conference, and to be awarded the best paper award for it was awesome, and a very rewarding experience overall. Additionally, I'd like to thank Univ.-Prof. DI Dr. Horst Bischof for allowing me to choose this specific topic as my thesis at ICG.

I'd also like to thank my colleagues that I shared an office with for the last ≈ 1.5 years, Christian, and Darko. Thank you for all the interesting discussions and teaching me all the basics of deep learning and research within that field. Especially Christian was a major help at all times. From teaching me the basics of deep learning to common python questions, there is almost nothing Christian could not help with.

I'd like to thank my family and friends, who provided constant support throughout all of my studies, from beginning, to end. Thank you to mom and dad, for always caring about me, providing support (emotionally and financially) and always reinforcing my strengths, enabling me to pursue whatever I wanted. Thank you to my brothers Michi & Georg as well as their wives Bettina & Michi for all the fun times, and it's great that I can always depend on you, whatever the situation might be.

I would not have had such a nice experience during my studies without my study-buddies, Thomas, Martin and Daniel. Thank you for making university fun, and let's hope for a lot more fun 'Stammtisch' meetings!

Thank you to all my friends that I spent my holidays in Croatia with every year for the last few years, it was great every year!

Additionally, I'd like to thank my oldest friend, Patrick. Thank you for distracting me from all the university stuff stuck in my head, I really enjoy our gaming sessions and that we can talk about absolutely anything, and I'm so glad that you are a part of my life.

Finally, I'd like to thank Martina. Thank you for keeping me sane throughout this journey - without you, this would not have been possible. Your constant love and support kept me motivated until the end, and thank you for always listening to my complaints about anything, it really helped a lot. I could not have met anyone better than you, thank you for everything, I love you.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions and Outline . . . 4

2 Deep Learning **5**
- 2.1 Artificial Neurons . . . 6
- 2.2 Feedforward Neural Networks . . . 7
- 2.3 Optimizing Neural Networks . . . 8
  - 2.3.1 Gradient Descent . . . 9
  - 2.3.2 Efficient Gradient Computation using Backpropagation . . . 11
  - 2.3.3 Stochastic Gradient Descent . . . 13
  - 2.3.4 Gradient Descent with Momentum . . . 13
  - 2.3.5 Adaptive Learning Rate Optimizers . . . 15
    - 2.3.5.1 Adagrad . . . 15
    - 2.3.5.2 RMSProp . . . 16
    - 2.3.5.3 Adam . . . 16
  - 2.3.6 Initialization of Weights and Biases . . . 17
  - 2.3.7 Regularization . . . 17
    - 2.3.7.1 Early Stopping . . . 18
    - 2.3.7.2 Weight Decay . . . 18
    - 2.3.7.3 Dropout . . . 19
  - 2.3.8 Batch Normalization . . . 20
- 2.4 Activation Functions . . . 21
  - 2.4.1 Sigmoid . . . 21
  - 2.4.2 Hyperbolic Tangent . . . 21
  - 2.4.3 Rectified Linear Unit . . . 23

2.4.4 Leaky Rectified Linear Unit 24
2.5 Convolutional Neural Networks 24
2.5.1 Convolution 25
2.5.1.1 Overview 25
2.5.1.2 Parameterization 27
2.5.2 Pooling 30
2.5.3 Fractionally Strided Convolution 30
2.6 Image Segmentation using Deep Convolutional Neural Networks 31
2.6.1 Image Segmentation 31
2.6.2 Encoder-Decoder Architecture 32
2.6.3 Softmax Cross-Entropy Loss 34
2.7 Data Augmentation 35
2.7.1 Additive Noise 35
2.7.2 Intensity Scaling 36
2.7.3 Intensity Shift 37
2.7.4 Random Flipping 37
2.7.5 Random Translation 37
2.7.6 Random Rotation 38
2.7.7 Random Cropping 39
2.7.8 Elastic Deformation 40
2.7.9 Synthetic Data Augmentation 41
2.8 Summary 41

3 Generative Adversarial Networks 43
3.1 Fundamentals 44
3.2 Deep Convolutional Generative Adversarial Networks 47
3.3 Wasserstein Generative Adversarial Networks 49
3.3.1 Wasserstein Generative Adversarial Networks with Gradient Penalty 51
3.4 SimGAN 53
3.5 Summary 56

4 Data Augmentation Using Generative Adversarial Networks 57
4.1 Modifying Generative Adversarial Networks for Data Augmentation 57
4.2 Implementation Details 59
4.2.1 Network Architectures: SCR Lung Database 60
4.2.2 Network Architectures: Cityscapes 62
4.3 Evaluation Metrics 64
4.3.1 Dice Coefficient 64
4.3.2 Intersection-over-Union 65
4.3.3 Hausdorff Distance 65

5 Evaluation and Results **67**
5.1 SCR Lung Database Medical Image Segmentation 67
5.1.1 Dataset Description 67
5.1.2 Evaluation Procedure 67
5.1.3 Results 69
5.1.4 Discussion 72
5.2 Cityscapes Semantic Image Segmentation 73
5.2.1 Dataset Description 73
5.2.2 Evaluation Procedure 75
5.2.3 Results 77
5.2.4 Discussion 77

6 Conclusion **83**

A List of Acronyms **85**

B List of Publications **87**
B.1 2017 87

Bibliography **89**

List of Figures

1.1 Image segmentation of the left lung of a lung X-ray image 3

2.1 Schematic of an artificial neuron 7
2.2 Structure of a simple Feedforward Neural Network (FNN) 8
2.3 Gradient descent applied on a 1-dimensional loss function 10
2.4 Gradient descent with momentum applied on a 1-dimensional loss function 14
2.5 Example of training and validation cost functions over iterations 19
2.6 Sigmoid activation 22
2.7 Hyperbolic Tangent (tanh) activation 22
2.8 Rectified Linear Unit (ReLU) activation 23
2.9 Leaky Rectified Linear Unit (leaky ReLU) activation 24
2.10 Two-dimensional convolution with a single kernel for a single-channel image 26
2.11 Two-dimensional convolution with a single kernel for an RGB image 26
2.12 Convolution on a one-dimensional input with different strides 28
2.13 One-dimensional convolution with zero-padding 28
2.14 Dilated convolution on a two-dimensional input for different dilations 29
2.15 Two-dimensional max-pooling for a single-channel image 31
2.16 Image segmentation of the left lung of a lung X-ray image 32
2.17 U-Net style encoder-decoder network architecture 33
2.18 Additive noise for data augmentation 36
2.19 Intensity scaling for data augmentation 36
2.20 Intensity shift for data augmentation 37
2.21 Horizontal flipping for data augmentation 38
2.22 Random translation for data augmentation 38
2.23 Random rotation for data augmentation 39
2.24 Random cropping for data augmentation 40

2.25 Elastic deformation for data augmentation 40

3.1 *GAN* architecture 44
3.2 *GAN* trained on MNIST 46
3.3 *GAN* trained on CIFAR10 46
3.4 Mode collapse in a *GAN* 47
3.5 Deep Convolutional Generative Adversarial Network (DCGAN) architecture 48
3.6 Samples from a *DCGAN* trained on bedroom images 49
3.7 Samples from a Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) trained on bedroom images 53
3.8 SimGAN architecture for unsupervised synthetic image refinement 54
3.9 SimGAN local adversarial loss 55
3.10 SimGAN history of refined images 55

4.1 *GAN* architecture for generation of image-segmentation pairs 58
4.2 Evaluation setup containing a pre-trained generator and U-Net style segmentation network 60

5.1 Example image and segmentation mask from the SCR Lung Database . . . 68
5.2 Raw output and thresholded segmentation mask sampled from the *GAN* trained on the SCR Lung Database 69
5.3 Sample images from our *GAN* trained on the SCR Lung Database 70
5.4 Comparison of good segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the SCR Lung Database 71
5.5 Comparison of the worst performing test sample from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the SCR Lung Database 71
5.6 Example image and category segmentation mask from the Cityscapes dataset 74
5.7 Example of a downsampled image and category segmentation mask from the Cityscapes dataset 76
5.8 Raw output and thresholded segmentation mask sampled from the *GAN* trained on the Cityscapes dataset 77
5.9 mean Intersection-over-Union (mIoU) for every category of the Cityscapes dataset 78
5.10 Sample images from our *GAN* trained on the Cityscapes dataset 79
5.11 Comparison of segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the Cityscapes dataset 81
5.12 Comparison of segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the Cityscapes dataset 82

List of Tables

4.1 Architecture of the *GAN* trained on the SCR Lung Database 61
4.2 Architecture of the segmentation network trained on the SCR Lung Database 62
4.3 Architecture of the *GAN* trained on the Cityscapes dataset 63
4.4 Architecture of the segmentation network used for the Cityscapes dataset . 64

5.1 Comparison of augmentation parameters for the SCR Lung Database 70
5.2 Segmentation performance comparison for the SCR Lung Database 72
5.3 Comparison of augmentation parameters for the Cityscapes dataset 75
5.4 Segmentation performance comparison for the Cityscapes dataset 78

1 Introduction

Contents

1.1 Motivation 1
1.2 Contributions and Outline 4

1.1 Motivation

Modern machine learning methods currently revolutionize our daily life. Covering a huge range of applications such as face detection [71], automatic speech recognition [24] and translation of text between different languages [70], and even highly advanced applications such as used in autonomous driving [14] and medical diagnosis [54], machine learning is already a major part of our lives. Especially *deep learning* [36] based methods consistently show improvements in the state-of-the-art every year, and for many *computer vision* tasks, even surpass human performance. However, what most of these methods have in common is that they are *supervised*, therefore requiring *annotated* data for training. Supervised methods try to predict a *label* y, given the input data x, therefore requiring a set of training data with corresponding annotation labels [18]. Contrary to supervised methods, *unsupervised* methods do not require labels, and most commonly try to learn more about the underlying structure or distribution of the data [18]. Furthermore, most deep learning methods require a large amount of data to train, often in the range of hundreds of thousands of images, only to be able to find a solution for a single task.

To circumvent the issue of insufficient annotated training data, there are three common approaches that can be taken when using deep learning for a new application where a large amount of specific, annotated data is not available. First, the most common approach is to use a public dataset containing images similar to the images required for the given task as training data. For certain applications, such as natural image classification, semantic segmentation of objects or urban areas, or digit classification, there are large, public datasets freely available. Additionally, for many applications and deep learning architectures, *pre-trained* network parameters for most of these public datasets are freely available, which can then be adjusted and fine-tuned to work on new datasets or tasks. This idea of using network parameters trained on a similar dataset and fine-tuning them to work on a different task is called *transfer learning*, and is based on the concept that early stages in a deep network capture useful representations, e.g. edge- or pattern-detection filters, that are useful even for different, related datasets [18]. Contrary to transfer learning, *domain adaptation* is a related concept that exploits the knowledge gained from pre-trained networks by fine-tuning on the same task (e.g. image classification with the same class definitions), but on related, different data [18]. However, even though transfer learning and domain adaptation are useful techniques in practice, as soon as the data required for any given task is more specific, and therefore very different to the data present in public datasets, those methods can not be applied easily.

The second common method of dealing with a low amount of data is to manually annotate additional data. While for some tasks, for example image classification, the labeling of data is not as expensive, especially in domains such as medical imaging, the acquisition of annotated groundtruth data is time-intensive, requires knowledge of experts, and is therefore very expensive. Furthermore, it is very difficult to predict how much data is required to train any given deep learning architecture, as the amount of required annotated images could vary by multiple orders of magnitude, depending on the task and application.

Finally, the third method to deal with a low amount of training data is to use *data augmentation*. Data augmentation is the process of generating additional training data from the available existing data. Typically, this is done by using annotation-preserving transformations on the input data, such as randomly rotating, translating or deforming the image. Through the random nature of data augmentation, it can be used to potentially generate an ‘infinite’ amount of training data by augmenting the already existing data. Although this is an effective way of dealing with the issue of low amounts of training data, it is not universally applicable, as the type of data augmentation and its parameters need to fit the intended task. For example, if the augmentation parameters are chosen such that they are too strong for a given task and deep learning setup, the augmentation might make the task too challenging to learn, which can significantly reduce the resulting performance. Furthermore, the parametrization of data augmentation methods introduces another set of very important hyperparameters, which can have a significant impact on the error made by the deep learning method.

While randomly augmenting data using simple transformations is the most common method of data augmentation, more sophisticated approaches for synthesizing additional training data have been proposed as well. However, since real and synthetic data distributions typically are very different, this is not an easy task, and requires additional post-processing on the synthetic data. For example, it has been shown that by rendering photorealistic, synthetic images and performing a set of transformations on those rendered images, they can be used to train an object detector with good performance [56]. Similarly, in the medical domain, it has been shown that by training a deep neural network on high-quality rendered 3D images from other computer vision tasks and fine-tuning it towards medical data, the general network performance can be improved when data is scarce [53]. This shows that data augmentation by using a generative model can improve the training of deep learning methods. For rendering-based methods, this generative model is typically carefully fine-tuned or matched to fit the real data distribution.

Recently introduced by Goodfellow et al. in 2014, Generative Adversarial Networks (GANs) provide an attractive method of automatically learning a generative model by just training a standard deep neural network [19]. A *GAN* consists of two subnetworks: the *generator*, and the *discriminator*, which are pitted against each other. The generator synthesizes data from an input noise vector. The discriminator is a standard classification network, which receives real data, as well as data from the generator as input. The goal of the discriminator is to perfectly classify each input image as either *real* or *synthetic*, while the goal of the generator is to synthesize images similar to the real data, fooling the discriminator. *GANs* have demonstrated potential in tasks such as state-of-the-art image generation ([21], [31]), domain-transfer ([30], [78]) or synthetic data generation ([64], [45]). However, while *GANs* show impressive results when trained on large datasets, it is still a topic of active research how *GANs* behave when trained on a small amount of data.

For this thesis, we will focus on data augmentation methods in the context of *image segmentation* tasks. Image segmentation is the process of partitioning an image into a set of regions that cover it [68]. This typically results in a segmentation mask, where every pixel has a label, and is therefore assigned to a specific region. Image segmentation is often used as a preprocessing step, to detect dense image regions of interest. Figure 1.1 illustrates an X-ray image and its corresponding segmentation mask of the left lung as an example.

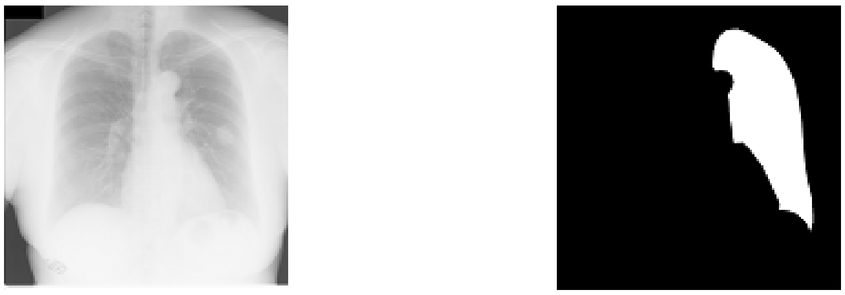

(a) Lung X-ray image. **(b)** Groundtruth segmentation mask for the left lung.

Figure 1.1: Image segmentation of the left lung of a lung X-ray image.

As segmentation is a pixel-wise problem, the acquisition of annotated segmentation masks is even more time consuming, compared to e.g. classification tasks, as a human annotator has to label every pixel manually. Therefore, an automated method of generating high quality synthetic data would resolve the issue of manual annotation, which would ideally lead to easier training of deep learning methods with an initially low amount of annotated data.

1.2 Contributions and Outline

The goal of this thesis is to evaluate the performance of *GANs* when used for data augmentation and to systematically study *GAN*-based augmentation and standard data augmentation. To achieve this, we introduce a novel modification to the standard *GAN* architecture, which simultaneously generates images and corresponding synthetic segmentation masks. These pairs of synthetic images and corresponding synthetic annotations allow us to train a deep neural network for image segmentation and evaluate the performance of *GAN*-based data augmentation in a quantitative manner. We evaluate our *GAN* architecture by comparing *GANs*-based augmentation to conventional data augmentation methods on two segmentation tasks, one from medical imaging, i.e. X-ray lung segmentation, and another, more challenging one from computer vision, i.e. urban scene understanding. Additionally, we compare the segmentation performance when training with different ratios of real and generated data, to further evaluate the impact of *GANs* samples on the training process.

Chapter 2 provides a thorough introduction to the topic of deep learning, and provides necessary background information for all further chapters. In Chapter 3, we discuss some of the most significant milestones in *GAN* research, ranging from the original *GAN*, as introduced by Goodfellow et al. in 2014 [19], to very recent advancements such as the Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) introduced by Gulrajani et al. in 2017 [21]. Additionally, we discuss the method of data augmentation called *SimGAN* [64] as an application closely related to our contribution. In Chapter 4, we describe our contribution and concrete implementation details of our deep convolutional networks for the *Lung SCR Database* [72] and the *Cityscapes* [10] dataset for semantic segmentation. Chapter 5 describes the evaluation setups for both datasets and presents all qualitative and quantitative results. Finally, Chapter 6 concludes this thesis, and illustrates possibilities for future investigation.

2 Deep Learning

Contents

2.1 Artificial Neurons 6
2.2 Feedforward Neural Networks 7
2.3 Optimizing Neural Networks 8
2.4 Activation Functions 21
2.5 Convolutional Neural Networks 24
2.6 Image Segmentation using Deep Convolutional Neural Networks 31
2.7 Data Augmentation 35
2.8 Summary 41

In the recent years, Deep Learning has made a significant impact on our everyday lives. Deep Learning is already used to automatically detect and recognize faces in our images [71], automatically analyze our speech [24], perform translation of text [70] and even segment medical image data to aid in medical diagnosis [54]. While conventional machine learning methods require careful tuning and data preprocessing by humans to extract meaningful representations out of raw input data, the key aspect of Deep Learning methods is to automatically learn representations and internal structure of raw input data, by using several stacked processing *layers* [36]. These layers typically compute simple differentiable functions on the input based on their parameters, which are called *weights*, and by feeding the input through multiple layers, more abstract representations can be learned. For example, in image classification, the output of the first few layers typically consists of edges at particular orientations and locations in the image, while later layers can encode more abstract concepts such as patterns or even objects by utilizing the information of the early layers. While the application of Deep Learning methods has turned out to be very successful for a wide variety of topics, we will mainly focus on the application of Deep Learning in the Computer Vision domain. In particular, we will focus on the problem of image segmentation, which, for example, is used in automatic object detection [62] or to localize and delineate anatomical structures and other regions of interest in medicine ([51], [50]). Section 2.1 will give an overview of the basic building blocks of artificial neural networks, the *artificial neurons*. Section 2.2 describes Feedforward Neural Networks (FNNs), which are a type of artificial neural network, in more detail. Section 2.3 describes the optimization process of neural networks, in particular the *backpropagation* algorithm, optimizers such as Stochastic Gradient Descent (SGD) [8] and regularization methods such as Dropout [67] and weight decay [35]. As activation functions of artificial neurons significantly impact gradients and training of neural networks, they are discussed in Section 2.4. In Section 2.5, the Convolutional Neural Network (CNN), which is a very popular extension of standard neural networks, is explained in more detail. As an application with specific interest to this thesis, image segmentation using deep *CNNs* is discussed in more detail in Section 2.6. Finally, Section 2.7 is dedicated to the process of data augmentation in deep learning, as this is the main topic investigated in this thesis.

2.1 Artificial Neurons

Artificial neurons are the basic building blocks of artificial neural networks. They describe a simple mathematical model inspired by neurons in the brain. The fundamental function of an artificial neuron is to receive multiple *inputs* $x_1, x_2, ...$ and compute a weighted sum z for these inputs using the *weights* $w_1, w_2, ...$ [47]. This weighted sum z is a *linear transformation* of the inputs of the neuron. Additionally, the *bias* b is added to the weighted sum of the inputs, and the result is passed through a non-linear *activation function* σ, resulting in the final output a. Thus, from a given set of inputs $x_1, x_2, ...$, the neuron performs a *nonlinear transformation* on its inputs by first computing a linear transformation, and additionally applying a bias and a nonlinear activation function, resulting in

$$a = \sigma\left(\sum_i x_i \cdot w_i + b\right) = \sigma\left(z\right), \tag{2.1}$$

which can be reformulated to a more convenient vector based notation exploiting the dot product of the vectorized weights $\boldsymbol{w}$ and the vectorized inputs $\boldsymbol{x}$

$$a = \sigma\left(\boldsymbol{x}^{\boldsymbol{T}} \cdot \boldsymbol{w} + b\right) = \sigma\left(z\right). \tag{2.2}$$

Figure 2.1 illustrates the structure of the artificial neuron. Through the choice of weights and bias, the artificial neuron can be *trained* to approximate a function given the inputs $\boldsymbol{x}$. In the earliest proposed artificial neurons, the Threshold Logic Units [43], the training parameters, which only consisted of weights and a threshold, therefore just computing a linear transformation, needed to be set manually. Nowadays, advanced automatic gradient-based learning methods are used, which are described in more detail in Section 2.3.

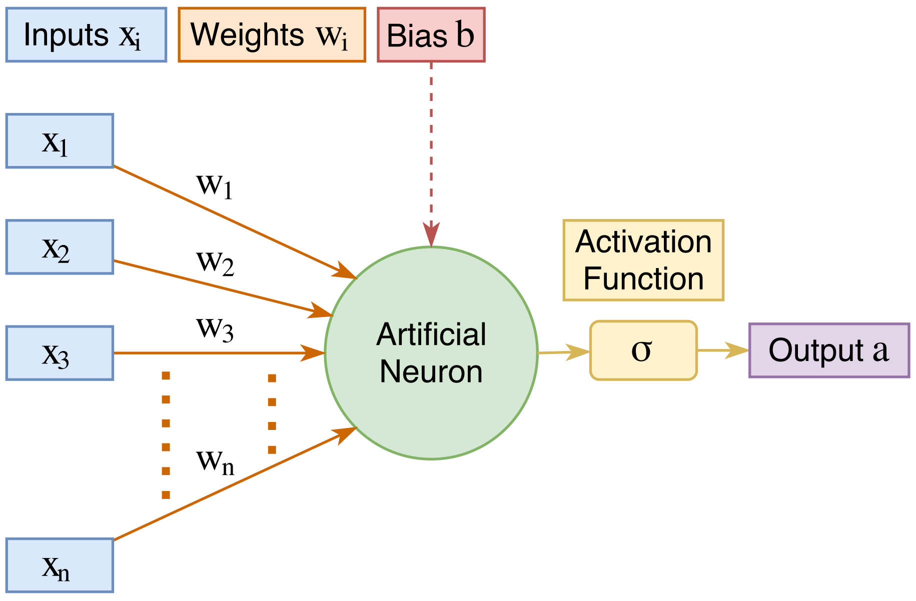


Figure 2.1: Schematic of an artificial neuron.

Although it is a very simple model, the artificial neuron has the capabilities to be used for a variety of problems. If the *step function* is chosen as the activation function, the neuron can be used for binary classification. These types of neurons used for binary classification are called *Perceptrons* [55]. In 1958, Rosenblatt proposed the first automatic learning algorithm for Perceptrons, that, given linearly separable training data, converges to a solution. This learning algorithm was based on adjusting the perceptron weights depending on the predicted class and the actual class from the training dataset, and therefore represents one of the earliest models for supervised learning. However, it was not until 1986, when Rumelhart et al. popularized the *backpropagation* algorithm for training neural networks [58], that combining artificial neurons to form neural networks was a focus of research. If a *logistic sigmoid* function is used as the activation function instead of a step function, this approach is commonly refered to as *logistic regression* [18]. By squashing the weighted sum of the inputs to the range between 0 and 1, the output can be interpreted as a probability. Therefore, in order to classify the outputs of logistic regression, it is necessary to threshold them. By combining multiple perceptrons into a network, it is even possible to solve problems that are not linearly separable, such as the *XOR problem* [18].

2.2 Feedforward Neural Networks

Feedforward Neural Networks (FNNs), similar to the previously discussed artificial neurons, have the goal of approximating a function given inputs $\boldsymbol{x}$ [18]. However, instead of being limited to very simple functions consisting only of a single weighted sum with an activation function, *FNNs* combine multiple neurons to form a directed graph without cycles. Artificial neurons at the same depth in this graph are grouped into *layers*, where each layer can be summarized as a single function consisting of these multiple neurons. An example of an *FNN* is illustrated in Figure 2.2. Every *FNN* consists of the inputs (commonly referred to as the *input layer*), an arbitrary number of intermediate layers of neurons called *hidden layers*, and a layer computing the outputs called the *output layer*. This layer based approach is also where the name *Deep Learning* comes from, as the *depth* of an *FNN* describes the number of layers an *FNN* consists of [18]. When every neuron in a layer is connected to all neurons in the following layer, this is called a *fully-connected* network, with layers exhibiting this behavior being called *fully-connected layers*. The simple *FNN* illustrated in Figure 2.2 is already much more powerful than a single artifical neuron. It can be shown that *FNNs* with a single hidden layer can be used to approximate any continuous function to any desired precision ([11], [27]). *FNNs* represent the

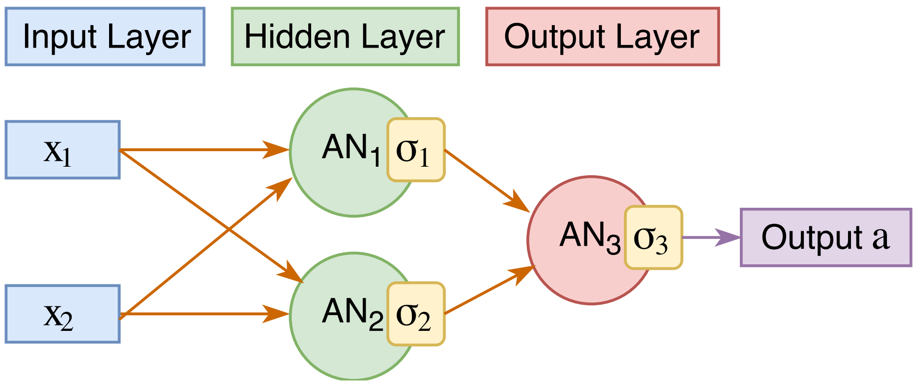


Figure 2.2: Structure of a simple *FNN*. For visual clarity, biases are omitted, even though every artificial neuron AN_i also has a bias term b_i.

foundation of most deep learning applications. The very popular and successful *CNNs*, which are described in more detail in Section 2.5, are simply an extension to standard *FNNs* [18]. In the following section we will show how, instead of manually tuning weights and biases of *FNNs* to approximate functions, modern methods automatically learn those network parameters from data.

2.3 Optimizing Neural Networks

The goal of optimizing *FNNs* is to automatically find weights and biases such that the network approximates the desired target output $\boldsymbol{y}$ given input $\boldsymbol{x}$. In order to achieve this, it is necessary to define a metric for how well the *FNN* approximates the output. This metric is commonly referred to as the *loss function* or *cost function* $J(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ describes the combined network parameters (weights, biases). Given a set of N *training* *examples* $\boldsymbol{x_T} = [\boldsymbol{x_{T_1}}, \boldsymbol{x_{T_2}}, ..., \boldsymbol{x_{T_N}}]$ and corresponding *targets* $\boldsymbol{y} = [\boldsymbol{y_1}, \boldsymbol{y_2}, ..., \boldsymbol{y_N}]$, $J(\boldsymbol{\theta})$ is typically computed as the average of the per-example loss function $L(\boldsymbol{a(x_{T_i};\theta)}, \boldsymbol{y_i})$, where $\boldsymbol{a(x_{T_i};\theta)}$ is the output of the *FNN*, given training example $\boldsymbol{x_{T_i}}$ as input and network parameters $\boldsymbol{\theta}$ [18]:

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{a(x_{T_i};\theta)}, \boldsymbol{y_i}) \tag{2.3}$$

For example, if L is defined as

$$L = ||\boldsymbol{a(x_{T_i};\theta)} - \boldsymbol{y_i}||^2 , \tag{2.4}$$

we arrive at the popular Mean Squared Error (MSE) loss function. A smaller loss function value is typically equal to a better function approximation of the *FNN*, therefore the training procedure of *FNNs* can be formulated as an optimization problem, where the goal is to minimize the loss function $J(\boldsymbol{\theta})$ with respect to the network parameters $\boldsymbol{\theta}$. This is commonly done using a variant of the *gradient descent* algorithm.

2.3.1 Gradient Descent

Given a real-valued loss function $J(\boldsymbol{\theta})$, such as described in the previous section, the goal of gradient descent is to find a *local minimum* of $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$. While for simple loss functions, it might be feasible to compute its minimum analytically, for more complex functions of multiple parameters, such as the loss functions of *FNNs* with millions of parameters, this quickly becomes infeasible [18]. In contrast to the analytical computation of minima, gradient descent is a numerical approach that works by choosing random starting parameters and repeatedly following the function towards the direction of its steepest descent. For a single training example $\boldsymbol{x_{T_i}}$ with the corresponding target output $\boldsymbol{y_i}$, the direction of steepest descent is given by computing the negative *gradient* of the per-example loss function with respect to the parameters $\boldsymbol{\theta}$ at the position $\boldsymbol{x_{T_i}}$ and $\boldsymbol{y_i}$:

$$-g_{\boldsymbol{\theta}_i} = -\nabla_{\boldsymbol{\theta}} L(\boldsymbol{a(x_{T_i};\theta)}, \boldsymbol{y_i}) \tag{2.5}$$

The final gradient for the loss function $J(\boldsymbol{\theta})$ is then given by computing the average of all gradients over the whole training set $\boldsymbol{x_T}$:

$$-g_{\boldsymbol{\theta}} = \frac{1}{N}\sum_{i=1}^{N} -g_{\boldsymbol{\theta}_i} \tag{2.6}$$

By defining a positive factor controlling the magnitude of the gradient descent, called the *learning rate* η, the gradient descent update rule of the *FNN* parameters $\boldsymbol{\theta}$ can be defined as

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \eta \cdot -g_{\boldsymbol{\theta}}. \tag{2.7}$$

An example of the gradient descent procedure applied on a simple 1-dimensional *MSE* loss is shown in Figure 2.3. The full gradient descent procedure is described in Algorithm 1.

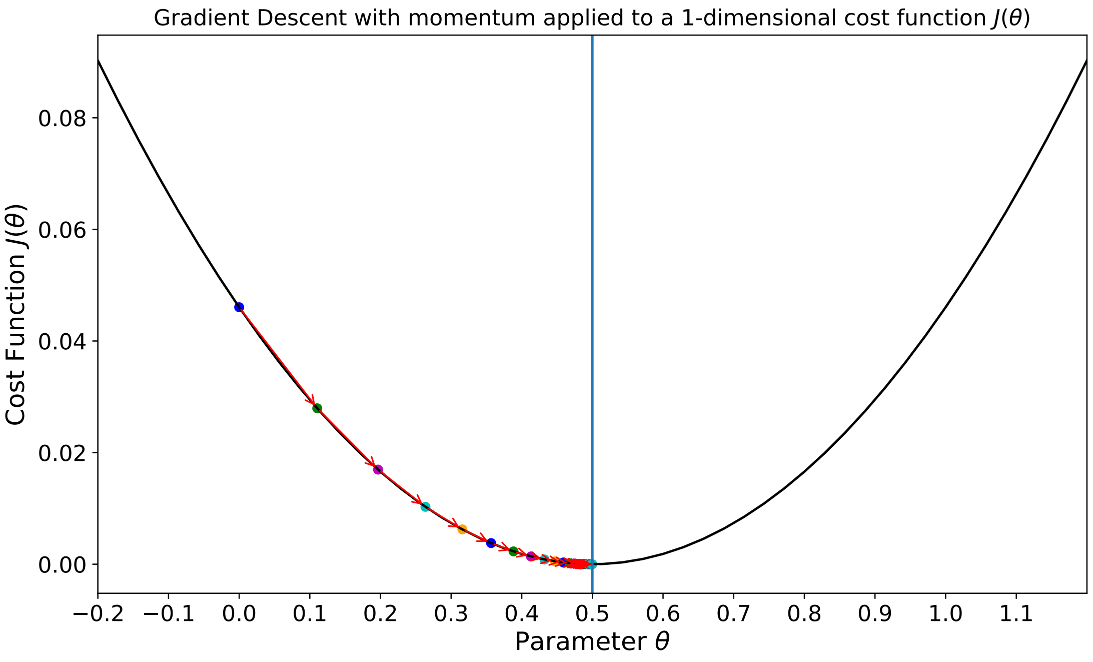


Figure 2.3: Gradient descent on a 1-dimensional loss function showing how the method iteratively gets closer to the global minimum. After just a few steps, gradient descent was able to find the optimum at the value of $\theta = 0.5$.

Depending on the initialization of the parameters, it is possible for gradient descent to find the global minimum of $J(\boldsymbol{\theta})$, however, this is not guaranteed, unless $J(\boldsymbol{\theta})$ is *convex*. Important to note is that for gradient descent to converge, the loss function needs to be smooth and provide gradients everywhere. This is also the reason why the chosen loss function is typically different from the actual objective, and a *surrogate* loss function is used instead [18]. As an example, instead of optimizing for the number of correctly classified examples in an image classification problem, we could optimize over the *MSE* of the predicted confidence and target confidences of each class. While the number of correctly classified examples would be discrete, and therefore non-smooth, the *MSE* would be smooth everywhere, providing useful gradients for gradient descent.

So far, we described gradient descent as an algorithm that can find a local minimum of a loss function $J(\boldsymbol{\theta})$ by computing gradients with respect to its parameters $\boldsymbol{\theta}$. However, the computation of this gradient with respect to all parameters of our *FNN* (weights, biases) is not straight forward - how can we determine the influence of a small change in a single weight or bias to the loss, and therefore the gradient of the loss with respect to this weight or bias? This is where the *backpropagation* [58] algorithm comes in.

Algorithm 1: Gradient descent algorithm.

```
Randomly initialize parameters $\boldsymbol{\theta} = \boldsymbol{\theta_0}$;
for number of iterations do
    for input training sample $\boldsymbol{x_{T_i}}$ do
        Compute the FNN output $a(\boldsymbol{x_{T_i}}; \boldsymbol{\theta})$ given the input $\boldsymbol{x_{T_i}}$ and the current
          parameters $\boldsymbol{\theta}$;
        Compute the gradient of the per-example loss function for the FNN output
          $a(\boldsymbol{x_{T_i}}; \boldsymbol{\theta})$ and the target output $\boldsymbol{y_i}$ with respect to the parameters $\boldsymbol{\theta}$:
            $g_{\boldsymbol{\theta}_i} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{a(x_{T_i}; \theta), y_i})$    (2.8)
    Update the parameters $\boldsymbol{\theta}$ by adding the negative average gradient, multiplied
      by the learning rate $\eta$, given the number of training examples $N$:
        $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \frac{1}{N} \sum_{i=1}^{N} g_{\boldsymbol{\theta}}$    (2.9)
```

2.3.2 Efficient Gradient Computation using Backpropagation

While the idea of backpropagation has been floating around since the 1970s [61], it was not utilized for neural network optimization until Werbos first applied it in 1981 [74], and Rumelhart et al. popularized it in 1986 [58]. The backpropagation algorithm is a procedure based on the chain rule of calculus to compute the gradient of the per-example cost function L with respect to each individual weight and bias ([47], [18]).

Before describing the algorithm, it is convenient to describe a matrix-based notation. The weight w^l_{jk} describes the weight connecting the k^{th} neuron in layer $(l-1)$ to the j^{th} neuron in layer l. From these weights, we can assemble a matrix $\boldsymbol{W}^l$, which is called the *weight matrix* for the l^{th} layer, where the entry in the j^{th} row and k^{th} column is equal to w^l_{jk}. Given K neurons in the layer $(l-1)$ and J neurons in the layer l, this matrix has the following form:

$$\boldsymbol{W}^l = \begin{bmatrix} w^l_{11} & \dots & w^l_{1K} \\ \vdots & \ddots & \vdots \\ w^l_{J1} & \dots & w^l_{JK} \end{bmatrix} \tag{2.10}$$

Similarly, the biases b^l_j, the pre-activation outputs z^l_j and the outputs a^l_j of the j^{th} neuron in layer l can be vectorized as $\boldsymbol{b}^l$, $\boldsymbol{z}^l$ and $\boldsymbol{a}^l$, respectively:

$$\boldsymbol{b}^l = \begin{bmatrix} b^l_1 \\ \vdots \\ b^l_J \end{bmatrix} \qquad \boldsymbol{z}^l = \begin{bmatrix} z^l_1 = \sum_k w^l_{jk} a^{(l-1)}_k + b^l_j \\ \vdots \\ z^l_J = \sum_k w^l_{jk} a^{(l-1)}_k + b^l_j \end{bmatrix} \qquad \boldsymbol{a}^l = \begin{bmatrix} a^l_1 = \sigma(z^l_1) \\ \vdots \\ a^l_J = \sigma(z^l_J) \end{bmatrix} \tag{2.11}$$

As the first step of the backpropagation algorithm, the *forward propagation* results are computed. During this step, the *FNN* receives a single training example $\boldsymbol{x}_{\boldsymbol{T_i}}$ as input and computes the outputs of each layer $\boldsymbol{a}^l$ as well as the pre-activation outputs of each layer $\boldsymbol{z}^l$, up to the final layer D. Additionally, given the final *FNN* output $\boldsymbol{a}^D$, the per-example loss $L(\boldsymbol{a}^D, \boldsymbol{y_i})$ is computed, given the target output $\boldsymbol{y_i}$ for this training example. This is called forward propagation, because the network outputs are iteratively computed for every layer, starting from the input layer, up to the output layer. Afterwards, the *backward propagation* is done by first computing the gradient $\boldsymbol{g}_{\boldsymbol{a}}^D$ on the output layer D, with respect to the *FNN* output:

$$\boldsymbol{g}_{\boldsymbol{a}}^D = \nabla_{\boldsymbol{a}^D} L(\boldsymbol{a}^D, \boldsymbol{y_i}) \tag{2.12}$$

Since we are interested in the gradient with respect to the pre-activation output, we compute that by simply performing an elementwise product $\odot$ (also known as Hadamard product) between the post-activation gradient and the derivative of the activation function:

$$\boldsymbol{g}_{\boldsymbol{z}}^D = \boldsymbol{g}_{\boldsymbol{a}}^D \odot \sigma'(\boldsymbol{z}^D) \tag{2.13}$$

This pre-activation gradient $\boldsymbol{g}_{\boldsymbol{z}}^D$ can then be used to compute the gradients of the cost function $J(\boldsymbol{\theta})$ with respect to the weight matrix $\boldsymbol{W}^D$ and the bias vector $\boldsymbol{b}^D$:

$$\boldsymbol{\nabla}_{\boldsymbol{b}^D} L(\boldsymbol{a}^D, \boldsymbol{y_i}) = \boldsymbol{g}_{\boldsymbol{z}}^D \tag{2.14}$$

$$\boldsymbol{\nabla}_{\boldsymbol{W}^D} L(\boldsymbol{a}^D, \boldsymbol{y_i}) = \boldsymbol{g}_{\boldsymbol{z}}^D (\boldsymbol{a}^{(D-1)})^\intercal \tag{2.15}$$

To arrive at the post-activation gradients of the next lower-level hidden layer $(D-1)$, we simply multiply the transposed weight matrix $(\boldsymbol{W}^D)^\intercal$ by the pre-activation gradient of the current layer $\boldsymbol{g}_{\boldsymbol{z}}^D$:

$$\boldsymbol{g}_{\boldsymbol{a}}^{(D-1)} = \nabla_{\boldsymbol{z}^{D-1}} L(\boldsymbol{a}^D, \boldsymbol{y_i}) = (\boldsymbol{W}^D)^\intercal \boldsymbol{g}_{\boldsymbol{z}}^D \tag{2.16}$$

The above steps can now iteratively be performed until every individual gradient with respect to every weight and bias in the first layer is known, therefore resulting in the final per-example gradient

$$g_{\boldsymbol{\theta}_i} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{a}(\boldsymbol{x}_{\boldsymbol{T_i}}; \boldsymbol{\theta}), \boldsymbol{y_i}), \tag{2.17}$$

for a given training example $\boldsymbol{x}_{\boldsymbol{T_i}}$ and target output $\boldsymbol{y_i}$. Therefore, using the backpropagation algorithm, we now have a method to compute the gradient for a single training example mentioned in Equation 2.8, which is required by gradient descent to update the *FNN* parameters ([47], [18]).

However, a big drawback of optimizing neural networks via backpropagation and gradient descent is the *vanishing gradient problem* [26]. As described above, the gradient for a specific weight is computed by backpropagating from the output towards the respective neuron. In this backpropagation path, through the usage of the chain-rule of calculus, gradients are repeatedly multiplied from back to front. Therefore, when gradients are small (which can happen e.g. when using saturating activation functions such as sigmoid or hyperbolic tangent), this can result in tiny gradients for the neurons in the front layers of the network, which results in slow learning for these neurons. While this is a problem with backpropagation and gradient descent itself, modern methods in deep learning mostly remedy this issue. Rectified Linear Unit (ReLU) [44] activations do not saturate the gradient magnitude in the positive input region, and therefore do not experience vanishing gradients in that region. Additionally, a recent type of network architecture, called Residual Network (ResNet) [23], bypasses neurons or layers with vanishing gradients by employing shortcut paths, allowing gradients to flow uncircumvented through much deeper networks. Finally, by normalizing intermediate layers using batch normalization [29], the distribution of inputs is more stable, resulting in a lower chance of getting stuck in saturated modes.

2.3.3 Stochastic Gradient Descent

While standard gradient descent using backpropagation is a useful method to train *FNNs* automatically, gradient descent itself requires a lot of computational resources when the training set is very large, as one single update step requires the computation of all gradients for all training examples.

The goal of Stochastic Gradient Descent (SGD) is to speed up the learning process by slightly modifying the standard procedure of gradient descent. The main difference between *SGD* and standard gradient descent is that *SGD* estimates the gradient of the cost function $\nabla_{\theta} J(\boldsymbol{\theta})$ by computing the gradients of the per-example loss only for a *small subset* of m randomly chosen training examples from the training set $\boldsymbol{x_M} = [\boldsymbol{x_{M_1}}, \boldsymbol{x_{M_2}}, ..., \boldsymbol{x_{M_m}}] \subset \boldsymbol{x_T}$, with corresponding target outputs $\boldsymbol{y_M} = [\boldsymbol{y_{M_1}}, \boldsymbol{y_{M_2}}, ..., \boldsymbol{y_{M_m}}] \subset \boldsymbol{y}$. This subset of training examples is called a *minibatch*, where m is the number of examples in this minibatch, called *minibatch size* [47]. The minibatches are chosen without replacement, i.e. every training example from the training set is chosen exactly once before a repeating example is chosen again. The gradient approximation using minibatches can be summarized as

$$\nabla_{\theta} J(\boldsymbol{\theta}) = \nabla_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{a}(\boldsymbol{x_{T_i}}; \boldsymbol{\theta}), \boldsymbol{y_i}) \approx \nabla_{\theta} \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{a}(\boldsymbol{x_{M_i}}; \boldsymbol{\theta}), \boldsymbol{y_{M_i}}). \tag{2.18}$$

For very small minibatch sizes, this only very roughly approximates the actual gradient, but in practice, *SGD* and algorithms derived from it have been shown to converge well [18].

2.3.4 Gradient Descent with Momentum

As described in Section 2.3.1, gradient descent can be used to find a local minimum of a function. However, depending on the shape of the function, the iterative approach of gradient descent can often take a large number of steps[1]. Especially for functions containing many almost flat regions with small gradients, traversing the function space with gradient descent is slow [69]. To solve this, *momentum* is added to the gradient descent algorithm. The main idea of momentum is to add short-term memory to gradient descent, also sometimes called acceleration. Effectively, this is a Taylor approximation of a second order scheme, keeping only first and second order terms (gradients and acceleration, respectively). We change the weight update step in Equation 2.9 to

$$\boldsymbol{g_\beta} = \beta\,\boldsymbol{g_\beta} + \frac{1}{N}\sum_{i=1}^{N} g_{\boldsymbol{\theta}_i} \tag{2.19}$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{g_\beta}, \tag{2.20}$$

where the initial $\boldsymbol{g_\beta}$ is set to zero, η is the learning rate and β is known as the momentum term. An example of momentum on a 1-dimensional *MSE* loss is shown in Figure 2.4.

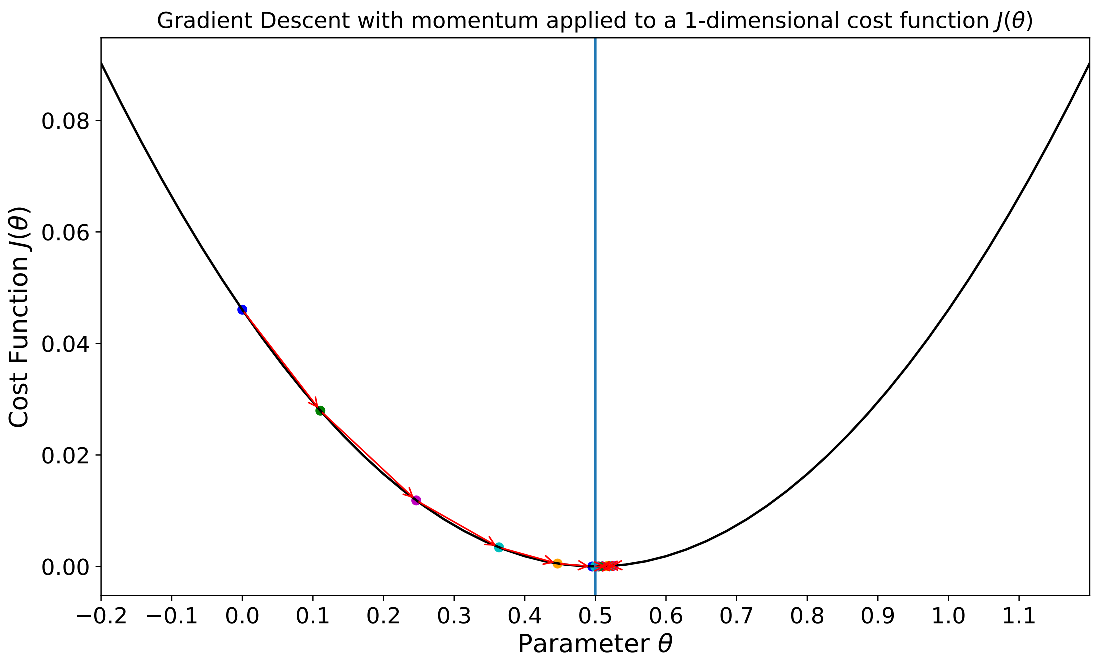


Figure 2.4: Gradient descent on a 1-dimensional loss function with momentum. The influence of the momentum term β can be seen in the faster initial convergence (compared to Figure 2.3) and small ‘overshoot’ over the optimum at $\theta = 0.5$.

Compared to the same setup shown in Figure 2.3 for gradient descent without momentum, it can clearly be seen that, for loss functions with very nice properties (single global

[1]Why Momentum Really Works, `https://distill.pub/2017/momentum`, Accessed: 01.02.2018

minimum, convex, smooth), gradient descent with momentum converges faster for the same learning rate.

2.3.5 Adaptive Learning Rate Optimizers

While *SGD* is a very powerful optimization method for training *FNNs*, it is not trivial to choose the best learning rate η for any given task. If η is chosen too large, the training might oscillate, not converge, or skip over relevant local minima. If it is chosen too small, it significantly delays the convergence process.

A common technique to circumvent this issue is to use *learning rate decay* [40]. For example, using *step decay*, the learning rate can be reduced by some factor every few epochs, which allows for a large learning rate at the beginning of training and a smaller learning rate towards the end of training. However, this decay procedure is also a hyperparameter in itself, and it needs to be designed carefully depending on the application.

Adaptive learning rate optimizers aim to work around the problem of finding the correct learning rate differently. In these methods, the learning rate η is not a global variable, but instead every trainable parameter now has a separate learning rate for itself. While these methods often still require some hyperparameter tuning, the main argument is that they work well for a broader range of setups, often when just using suggested default hyperparameters [40]. This section summarizes commonly used adaptive learning rate optimizers Adagrad [13], RMSProp [25], and Adaptive Moment Estimation (Adam) [32] for training deep *FNNs*.

2.3.5.1 Adagrad

The main idea of *Adagrad* [13] is to keep track of the sum of squared gradients for each parameter, and use that sum to normalize the parameter update element-wise [40]. This is illustrated in the following adjusted parameter update when using minibatch gradient descent

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\boldsymbol{G} + \epsilon}} \odot \frac{1}{m} \sum_{i=1}^{m} g_{\boldsymbol{\theta}_i}, \tag{2.21}$$

where the additional normalization term $\sqrt{\boldsymbol{G} + \epsilon}$ consists of the matrix $\boldsymbol{G}$ containing the sum of squares of the past gradients with respect to all parameters $\boldsymbol{\theta}$ along its diagonal, and ϵ is a term to avoid divison by zero [57]. By using this per-parameter learning rate, Adagrad effectively eliminates the need to tune the learning rate. However, since the squared gradients are strictly accumulated in the matrix $\boldsymbol{G}$, and therefore the accumulated sum is strictly growing, the effective learning rate shrinks during training until it eventually becomes too small to provide any useful parameter update.

2.3.5.2 RMSProp

In order to reduce the aggressive, monotonically decreasing learning rates provided by Adagrad, RMSProp [25] adjusts the matrix $\boldsymbol{G}$ from the Adagrad update to not contain the accumulated sum of squared gradients of all parameters in its diagonals. Instead, it uses a moving average of squared gradients in its diagonals given as

$$G_{ii} = \gamma G_{ii} + (1-\gamma)\left(\frac{1}{m}\sum_{i=1}^{m} g_{\boldsymbol{\theta_i}}\right)_i^2 . \tag{2.22}$$

The *decay rate* γ is now an additional hyperparameter, with a proposed default value of 0.9. The main parameter update is the same as for Adagrad, therefore this method still adjusts the learning rate of each parameter based on the magnitude of its gradients, however, the moving average prevents updates from getting monotonically smaller [40].

2.3.5.3 Adam

Similarly to how *SGD* was extended with momentum, Adaptive Moment Estimation (Adam) [32] extends RMSProp by introducing an exponentially decaying average of past gradients

$$\boldsymbol{\mu} = \beta_1 \boldsymbol{\mu} + (1-\beta_1)\left(\frac{1}{m}\sum_{i=1}^{m} g_{\boldsymbol{\theta_i}}\right) \tag{2.23}$$

$$\boldsymbol{v} = \beta_2 \boldsymbol{v} + (1-\beta_2)\left(\frac{1}{m}\sum_{i=1}^{m} g_{\boldsymbol{\theta_i}}\right)^2 , \tag{2.24}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{v}$ are the estimates of the first and second moment of the gradients, respectively [57]. As $\boldsymbol{\mu}$ and $\boldsymbol{v}$ are initialized as zero-vectors, Kingma and Ba [32] observed that they are initially biased towards zero. By using bias-corrected estimates

$$\hat{\boldsymbol{\mu}} = \frac{\boldsymbol{\mu}}{1-\beta_1^t} \tag{2.25}$$

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v}}{1-\beta_2^t}, \tag{2.26}$$

where t is the current training iteration number, they circumvent this issue which leads to an update rule similar to Adagrad and RMSProp:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon}\hat{\boldsymbol{\mu}} \tag{2.27}$$

Recommended values by the authors for the parameters of *Adam* are $\epsilon = 1 \cdot 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

2.3.6 Initialization of Weights and Biases

While describing the gradient descent algorithm in Section 2.3.1, we mentioned that gradient descent works by iteratively stepping towards the direction of steepest descent, starting from an initial position. For neural networks, this initialization needs to be chosen very carefully. For example, if multiple neurons in the same hidden layer share the same weights, they will receive the same gradients, and therefore compute the same results, leading to wasted model capacity [6]. Typically, neural network weights are simply initialized from a zero-mean Gaussian destribution with a small standard deviation [34]. A problem with this, however, is that the variance of the distribution of outputs from a randomly initialized neuron grows with the number of inputs [40]. To normalize the variance of each neuron's output to 1, it is sufficient to use a standard normal distribution and scale the weight by the square root of its *fan-in* n_{in}, which is the number of its inputs:

$$\boldsymbol{w}_0 \sim \frac{\mathcal{N}(0,\,1)}{\sqrt{n_{in}}} \tag{2.28}$$

Similarly, Glorot and Bengio performed an analysis on the backpropagated gradients and recommend an initialization [16] (known as *Xavier* or *Glorot* initialization) of

$$\boldsymbol{w}_0 \sim \sqrt{\frac{2}{n_{in}+n_{out}}}\,\mathcal{N}(0,\,1), \tag{2.29}$$

where n_{out} is the *fan-out*, describing the number of output units. Specifically for neurons with *ReLU* activation, He et al. recommend using an initialization [22] (known as *He* intialization) of

$$\boldsymbol{w}_0 \sim \sqrt{\frac{2}{n_{in}}}\,\mathcal{N}(0,\,1). \tag{2.30}$$

For biases, it is common to simply initialize all biases to 0 in *FNNs* [40].

2.3.7 Regularization

So far, we only mentioned training an *FNN* with gradient descent and backpropagation, using the training set $\boldsymbol{x_T}$ with the corresponding labels $\boldsymbol{y}$. While this allows us to train our *FNNs* to predict outputs for our training set, it does not necessarily mean that it is able to predict outputs correctly for unseen data. Therefore, typically two additional sets of data are introduced to optimize *FNNs*, the *validation set* $\boldsymbol{x_{val}}$ and the *test set* $\boldsymbol{x_{test}}$. All three sets (training, validation, test) are disjoint, such that no sample is common among them. The validation set is typically used to fine-tune the *FNN* model hyperparameters, such as the network architecture or the learning rate. The test set is only used for the final evaluation to check the performance of the *FNN* on previously unseen data. If an *FNN* does not generalize well, i.e. has a smaller training loss than test loss, this is called *overfitting*, while the inverted scenario, i.e. it has a much smaller test loss compared to the training loss, is called *underfitting*. Typically, overfitting and underfitting are directly related to the *model capacity* of a machine learning method. Loosely speaking, the model capacity of a deep network is directly correlated to the number of parameters inside the network. The model capacity decides how well a deep network is able to fit a wide variety of functions [18]. If the capacity is too small, the network might be unable to fit the training set (underfitting), while a model capacity that is too large might lead to memorization of training samples (overfitting) [18]. Underfitting is typically not much of an issue with *FNNs*, as this can mostly be remedied by using a more powerful or deeper network architecture with more parameters. Overfitting, however, needs to be taken into account to be able to use *FNNs* for new, unseen data. The process of reducing the effect of overfitting or preventing it altogether is called *regularization* [18]. This section briefly describes the most popular regularization techniques for *FNNs*.

2.3.7.1 Early Stopping

When the model capacity of an *FNN* is large enough to be able to overfit, it is typically observed that the training loss decreases steadily until convergence, while the validation loss decreases at the start and rises again after the *FNN* starts to overfit. *Early stopping* aims to regularize the *FNN* by finding the network parameters at the point of the lowest validation loss. By using those network parameters with the lowest validation loss, the network potentially generalizes better to unseen data. Therefore, early stopping basically prevents overfitting before it has a measurable impact on the validation loss. An example where the benefit of early stopping can be seen is illustrated in Figure 2.5.

2.3.7.2 Weight Decay

Weight decay, also known as L^2 parameter regularization, is a strategy that adds a regularization term to the loss function which penalizes large weights:

$$\Omega(\boldsymbol{\theta}) = \alpha \frac{1}{2} ||\boldsymbol{w}||_2^2 . \tag{2.31}$$

The parameter α determines the amount of weight decay [18]. This effectively changes the gradient descent update for the network weights to

$$\boldsymbol{w} = (1 - \eta\alpha)\boldsymbol{w} - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{w}} L(\boldsymbol{a}(\boldsymbol{x}_{\boldsymbol{T_i}}; \boldsymbol{w}), \boldsymbol{y_i}). \tag{2.32}$$

This has the effect of only preserving parameters which significantly contribute to reducing the loss function, while decaying away the other parameters [18].

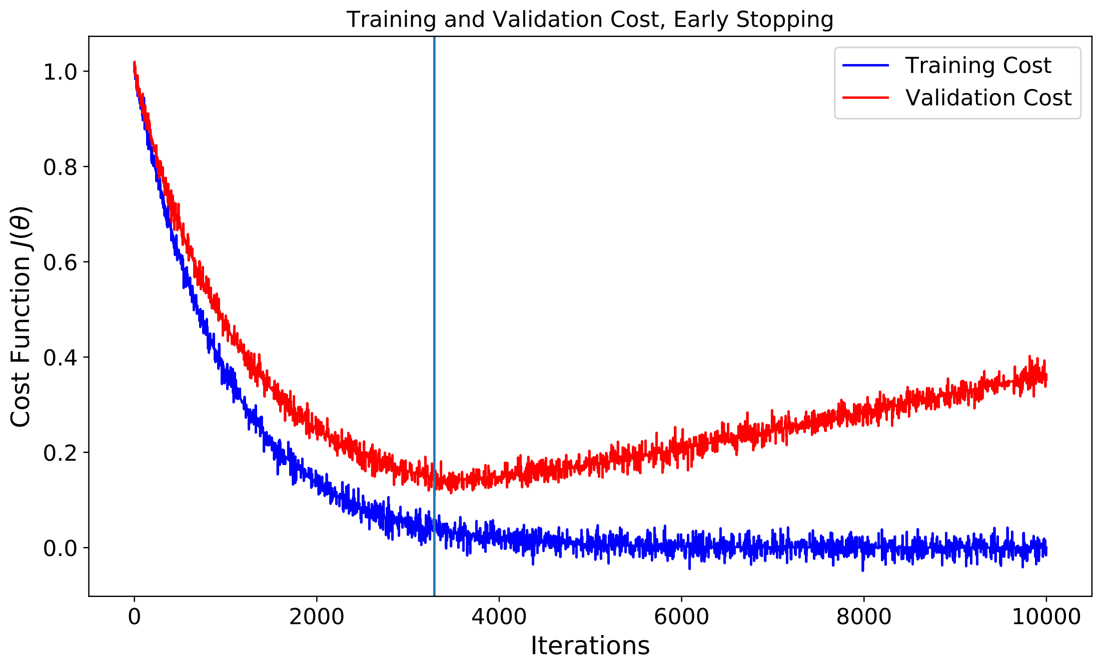


Figure 2.5: Example of possible training and validation cost functions over the iterations during training. Around iteration 3400, the network starts to overfit, resulting in increased validation cost while still reducing training cost over time. The vertical line marks the minimum of the validation cost, for which the model should provide the best generalization and therefore test performance.

2.3.7.3 Dropout

Dropout [67] is an efficient and powerful method of regularizing *FNNs*, which works by training a virtual ensemble of all subnetworks that can be formed by removing units not in the output layer from a base network. By multiplying its output value by zero, it is possible to effectively remove an artificial neuron from an *FNN* [18]. Therefore, a binary mask signifying if an artificial neuron is enabled or not can be defined. This mask is applied to all input and hidden neurons in the network for every example in a minibatch using a minibatch-based optimizer such as *SGD*. Each artificial neuron samples from this binary mask using a fixed probability called the *dropout rate*, which commonly is set to 0.5 for hidden neurons and 0.8 for input neurons, describing the probability of including that specific neuron. This means that for every training example, each neuron has a chance to be active, equal to the dropout rate. This forces the *FNN* to not depend on specific artificial neurons only, but to instead distribute the computation to multiple neurons, therefore reducing the impact of overfitting. For testing, the network weights are typically just divided by the inclusion probability for each neuron, which is called the weight scaling rule ([18], [67]).

2.3.8 Batch Normalization

Motivated by stability and performance issues when training very deep *FNNs*, Ioffe and Szegedy introduced *batch normalization* [29] as a method of adaptive reparamerization to better coordinate updates between many layers in a deep network [18]. During training, it is adventageous for every layer to have a fixed distribution, such that the following layer does not have to readjust its weights to acommodate for changes in distribution. For the input layer, this is commonly done by normalizing all training, validation, and test examples. The goal of batch normalization is to normalize intermediate layers during training as well [29]. For any given input or hidden layer, let $\boldsymbol{A}$ be a matrix containing a minibatch of post-activation outputs of that layer, where each row of $\boldsymbol{A}$ contains the outputs for its respective training example. In order to normalize $\boldsymbol{A}$ to have zero mean and unit variance, its elements are replaced with

$$A'_{i,j} = \frac{A_{i,j} - \mu_j}{\sigma_j}, \tag{2.33}$$

where μ_j and σ_j describe the mean and standard deviation of the post-activation output of neuron j, respectively, and i is the minibatch index. For the rest of the network, these adjusted outputs $\boldsymbol{A}'$ are used just the same way that the $\boldsymbol{A}$ was used originally. Most importantly, the computation of the mean and standard deviation as well as the normalization using them is part of the *FNN*, and therefore when computing the gradient of this neuron, the backpropagation algorithm will backpropagate through these computations. This results in removing the effect of operations that simply increase the mean or standard deviation in a neurons output [18]. Previously, normalization approaches either added additional penalty terms to force normalized output statistics, which led to imperfect normalization, or renormalized all neuron statistics after each gradient step, which was usually inefficient due to the required time. Batch normalization defines the respective units to be normalized by definition, therefore circumventing these issues. At training time, the mean and standard deviation of each neuron output is simply computed over the training minibatch. At test time, these definitions may be replaced by running averages collected during training.

The normalization of mean and standard deviation of a neuron output can reduce the expressive power of a *FNN* containing this neuron [18]. Therefore, it is common to replace the outputs $\boldsymbol{A}$ with $\boldsymbol{\gamma}\boldsymbol{A}' + \boldsymbol{\beta}$ rather than just the normalized $\boldsymbol{A}'$, where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned parameters that allow for the new formulation to contain any mean and standard deviation. This new parameterization can represent the same family of functions as the original parameterization, while at the same time being easier to learn with gradient descent [18]. Through batch normalization, both additive and multiplicative noise is introduced at training time, which can have a regularizing effect, sometimes even making other techniques such as Dropout unnecessary [18].

2.4 Activation Functions

As described in Section 2.1, artificial neurons typically use non-linear activation functions on the weighted sum of their inputs. This section describes commonly used activation functions as well as their properties during optimization.

2.4.1 Sigmoid

The sigmoid activation function has the mathematical form of

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.34}$$

which maps the real-valued input x to the range between 0 and 1 ([40],[18]). The sigmoid activation function is shown in Figure 2.6. In early neural networks, sigmoids were a popular choice due to the biologically inspired interpretation, as the output range between 0 and 1 could be interpreted as the firing rate of the neuron, where 0 represents a neuron that does not fire at all, and 1 represents a neuron firing at maximum frequency [40]. However, the sigmoid activation function has some significant drawbacks, resulting in other activation functions becoming more popular. The main drawback of sigmoids is that they saturate and therefore provide only gradients very close to zero in these regions, effectively preventing the backpropagation algorithm from providing gradients through this neuron to all the inputs ([40],[18]). Additionally, the output of the sigmoid activation is not zero-centered, which can lead to undesirable dynamics during gradient descent [40]. Therefore, using sigmoid activation functions for hidden neurons is discouraged [18]. For output neurons, the range between 0 and 1 can be useful, e.g. for interpreting predictions as probabilties.

2.4.2 Hyperbolic Tangent

The Hyperbolic Tangent (tanh) activation function is closely related to the sigmoid activation function and has the mathematical form of

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = 2\sigma(2x) - 1. \tag{2.35}$$

As can be seen in Equation 2.35, *tanh* is simply a scaled version of the sigmoid activation [40]. However, it is centered around 0, and therefore does not exhibit some of the issues that sigmoid activations have. As a result, *tanh* activations are almost always preferred to sigmoid activations ([40],[18]). The *tanh* activation is shown in Figure 2.7.

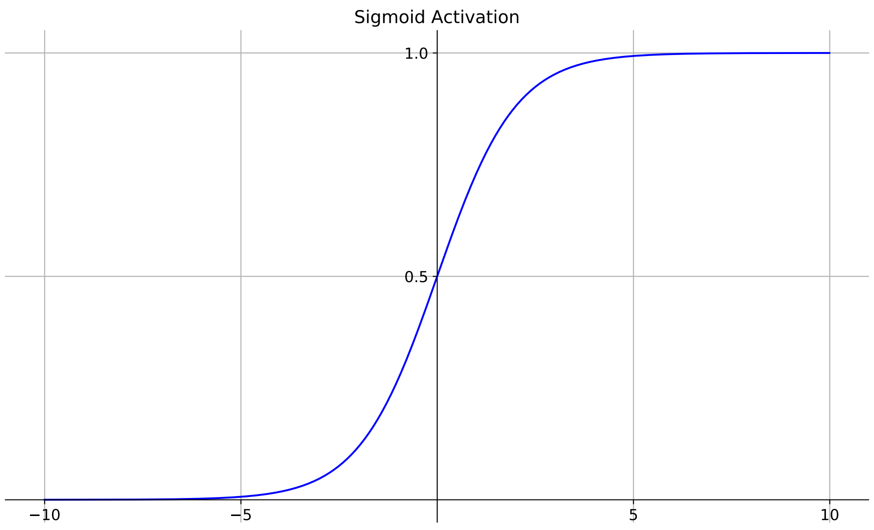


Figure 2.6: Sigmoid Activation function in the range of $x = [-10, 10]$.

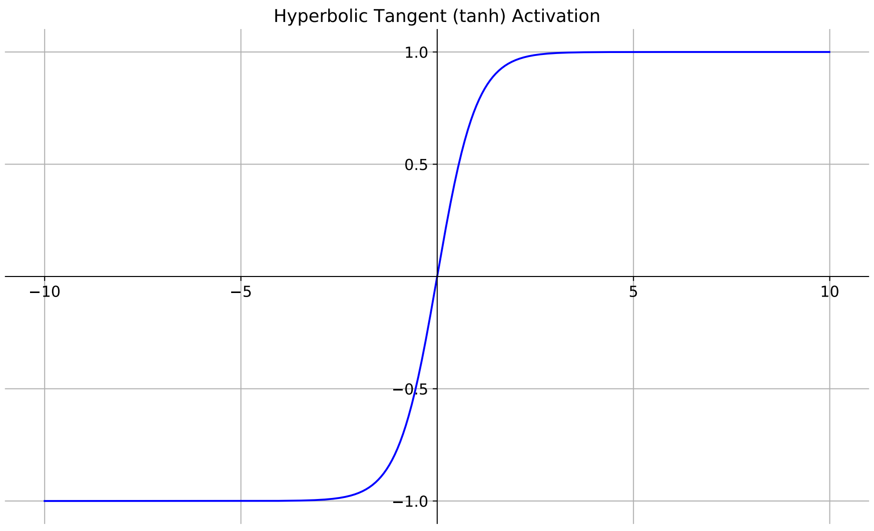


Figure 2.7: *tanh* Activation function in the range of $x = [-10, 10]$.

2.4.3 Rectified Linear Unit

The *ReLU* [44] activation is defined as

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.36}$$

which is illustrated in Figure 2.8. It was found that it significantly accelerates the convergence of *SGD* compared to sigmoid or *tanh* activations [34]. This is most likely due to the linear, non-saturating form of *ReLUs*, preventing the main issue of sigmoid units, the vanishing gradient problem ([26], [40]). It is also very efficient to compute, as it just requires thresholding the neuron activations at zero. Additionally, as *ReLUs* have a second derivative of 0 almost everywhere and a derivative of 1 in the active regions, the gradients are large and consistent in the active region, which is very useful for learning [18]. One of the main drawbacks of *ReLUs* is that units can 'die' during training. If the weights during training are shifted in such a way that for all data inputs during *SGD*, the neuron activation lies on the flat plane in the negative x-area, the neuron will only backpropagate a gradient of 0, making it very unlikely that this neuron will recover from this state [40]. This can especially occur if the learning rate is set too high, resulting in larger swings for parameter updates.

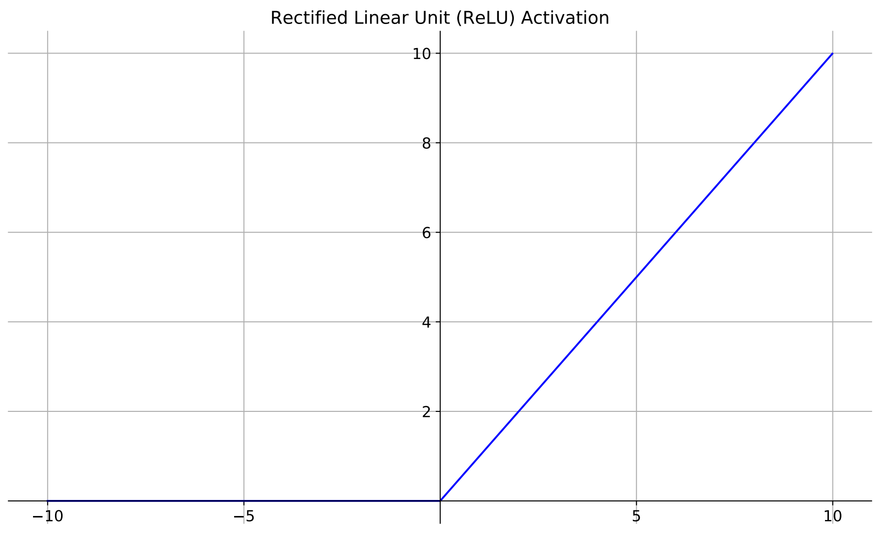


Figure 2.8: *ReLU* Activation function in the range of $x = [-10, 10]$.

2.4.4 Leaky Rectified Linear Unit

As a method to fix the problem of ‘dying’ *ReLUs*, the Leaky Rectified Linear Unit (leaky ReLU) [22] activation has been introduced, which has the mathematical form of

$$\text{leaky ReLU}(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases}, \tag{2.37}$$

where α is a small coefficient determining the negative slope of the function. The *leaky ReLU* activation is illustrated in Figure 2.9. Given that *leaky ReLUs* now provide gradients also for negative inputs, gradients are also non-zero in these regions, which prevents the neuron from ‘dying’ [40].

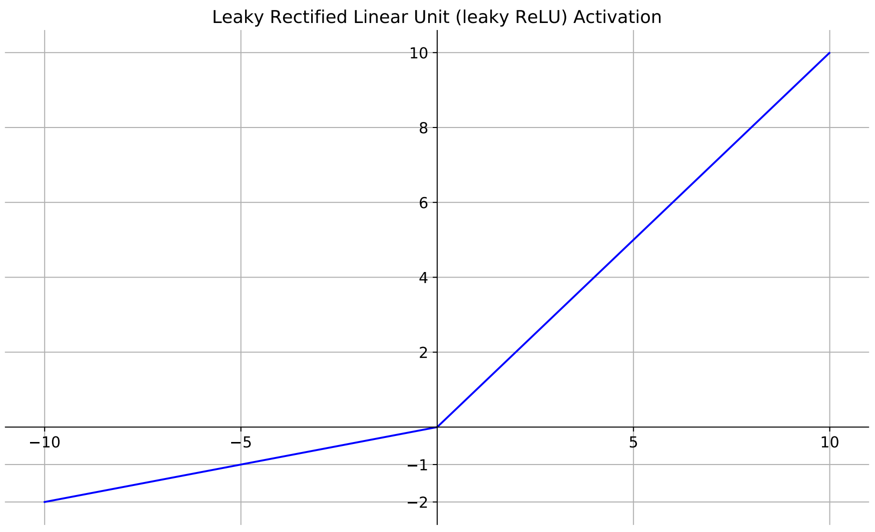


Figure 2.9: *leaky ReLU* Activation function in the range of $x = [-10, 10]$, with the slope coefficient set to $\alpha = 0.2$.

2.5 Convolutional Neural Networks

Up until this section, all neural networks we described were fully-connected *FNNs*, where every neuron in each layer except the input layer is connected to every neuron in the previous layer. Every neuron then computes a function of its inputs and its weights and biases, independently from all other neurons in the same layer. Consider a fully-connected *FNN* taking a simple RGB image (defined as a volume) of size $[256 \times 256 \times 3]$ as its input. Flattening this volume, every single neuron has $[256 \times 256 \times 3] = 196608$ weights, and that is only for a single neuron, while deep architectures require a lot of neurons and hidden layers to sufficiently represent complex structures contained in the input data.

This means that the fully-connected nature of these networks consumes a lot of memory, especially for large images or videos, and the huge number of parameters can lead to overfitting [40]. In order to address these issues, *CNNs* [37] were introduced as a very popular extension of standard neural networks. By definition, *CNNs* are simply neural networks that use *convolution* instead of matrix multiplication in at least one of their layers [18]. By exploiting the knowledge that neural network inputs commonly have a known, grid-like topology [18], such as in the case of time-series data, images, or volumes, this convolution operation allows for a much more efficient forward propagation pass that is very well suited for fast GPU implementation, in addition to vastly reducing the number of parameters in the network, which allows for increased network sizes [40]. Most importantly, everything regarding the optimization of standard fully-connected neural networks, such as the optimization using *SGD* and backpropagation, or the application of non-linear activation functions on the output still applies for *CNNs*.

2.5.1 Convolution

2.5.1.1 Overview

For a two-dimensional image I, the discrete convolution is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n), \tag{2.38}$$

where $K(m,n)$ is a two-dimensional *kernel*, and the output $S(i,j)$ is referred to as the *feature map* [18]. Intuitively, this operation 'slides' the kernel along the image I and computes a weighted sum of the kernel and the image at each position i, j. An example of such a discrete two-dimensional convolution is illustrated in Figure 2.10. The convolution operation can easily be extended to an arbitrary number of image and kernel dimensions. In the context of *CNNs*, the kernel K describes the learnable weights of the *convolutional layer*, and each convolutional layer can contain an arbitrary number of kernels, each resulting in its own output feature map, stacked along the feature 'depth' axis. Important to note is that the convolution is always performed along the full feature depth of the input, for example, if the input image was an RGB image with respective dimensions of $[8 \times 8 \times 3]$ (3 'slices', one for each color channel), the kernel would also need to be of size 3 along the depth axis, for example $[4 \times 4 \times 3]$. This convolution along the feature depth axis for an RGB image is visualized in Figure 2.11.

Typically, the kernel is much smaller than the image, resulting in *sparse connectivity* between inputs and outputs of a convolutional layer, which is one of the key motivations of *CNNs* [18]. Intuitively, the idea behind sparse connectivity in *CNNs* is that small kernels are good enough to detect useful features or patterns, such as edges, in images. Sparse connectivity also implicitly defines the *receptive field* of the output feature map entries. The receptive field in each spatial dimension is defined as the number of inputs each respective feature map output is affected by during the convolution [18]. For the

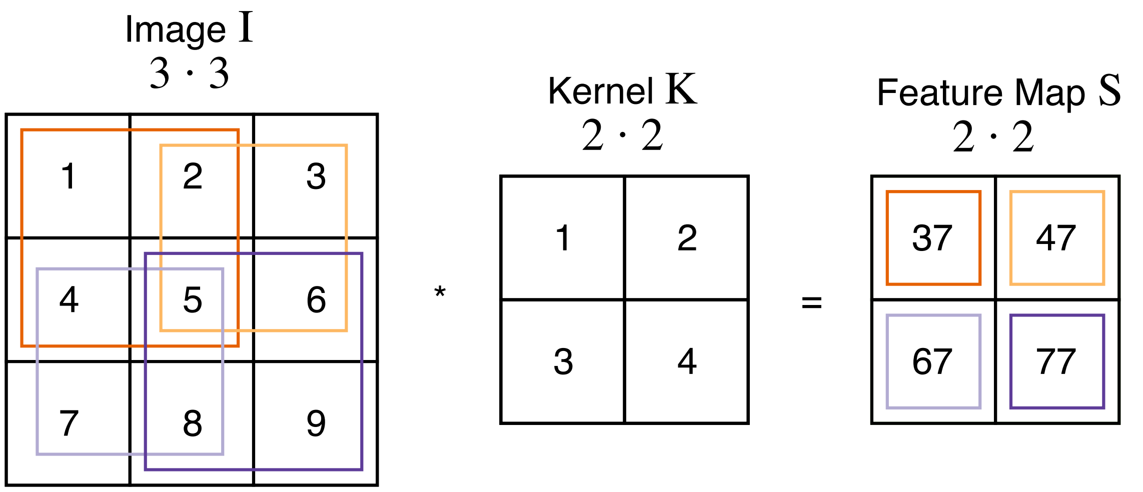


Figure 2.10: Two-dimensional convolution, where the kernel K is restricted to be strictly inside the image I. Every output element of the feature map S is computed as a weighted sum between the image I and the kernel K at a specific position i, j. The receptive field of each feature map entry is illustrated by correspondingly colored boxes in the image I.

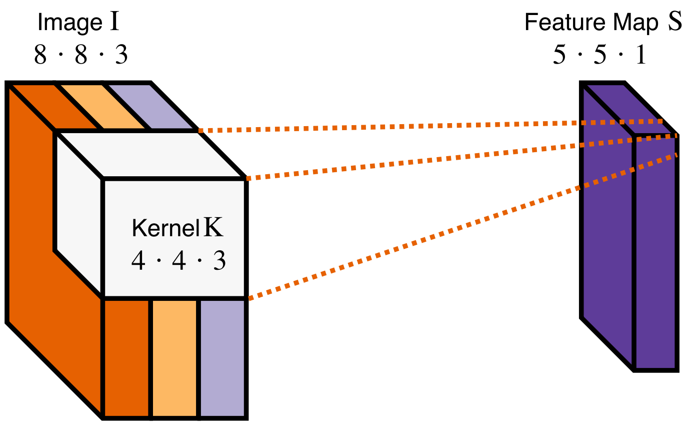


Figure 2.11: Two-dimensional convolution, where the kernel K is restricted to be strictly inside the RGB image I. The kernel has to accomodate for all the channels inside the input, therefore it needs to be of size 3 along the depth axis.

example shown in Figure 2.10, the receptive field is 2 in each spatial dimension, and is illustrated by the differently colored boxes, while Figure 2.11 demonstrates a receptive field of 4 in each spatial dimension.

In addition to sparse connectivity, *parameter sharing* is also a key motivation of *CNNs*. In traditional fully-connected neural networks, every element of the weight matrix is used exactly once when computing the output, as every element describes a unique weight between one input and one output [18]. For a *CNN*, each kernel entry is used at multiple positions across the image, therefore implicitly sharing that parameter across the whole image. While this does not affect the runtime performance of forward propagation compared to fully-connected networks, it again significantly reduces the memory footprint of *CNNs*. Parameter sharing also has a very useful side-effect: as the kernel keeps the same parameters at every position, the convolutional layer is invariant to translation of the input, as this will just simply translate the output as well.

2.5.1.2 Parameterization

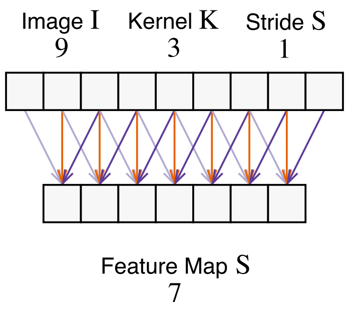
Each convolutional layer has a certain set of hyperparameters, each of which determine the receptive field, the number of connections and the output size of the feature maps.

Kernel Size The *kernel size* (sometimes also called *filter size*) K describes the receptive field, and therefore the size of the convolutional kernel that is applied to all input locations. Increasing this parameter allows the convolutional layer to pick up more spatial information, while simultaneously increasing the number of network weights. As previously discussed, the number of feature depth channels D of a convolutional kernel is always equal to the number of depth channels of the input [40].

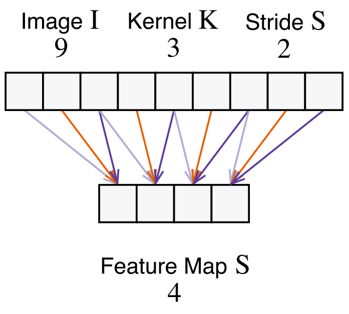
Number of Kernels The *number of kernels* directly corresponds to the number of learnable parameters and the depth D of the output volume of a convolutional layer. As each kernel produces an individual output feature map, D kernels produce an output feature map of depth D [40].

Stride As described previously, convolution can be understood as weighted summation by 'sliding' a kernel over an input volume. However, the 'sliding' does not need to happen with an offset of one pixel at a time, which is what the *stride* describes. The stride S specifies the number of pixels the kernel is moved between every output feature computation. Larger strides will produce smaller output feature maps, as a lower amount of computations (with larger distances in between) will be performed [40]. This concept is illustrated in Figure 2.12.

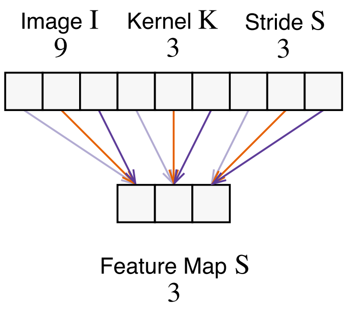
Zero-Padding Due to the way the convolution operation works, the spatial output dimensions of a convolutional layer shrink if the convolutional kernel is strictly enforced to be inside the image (compare to Figure 2.10 and Figure 2.11). However, it is often convenient to keep spatial dimensions of input and output layers the same, for example when computing functions in a pixel-wise manner. This is where *zero-padding* can be applied. By appending the input volume with zeros around the border, it is possible to circumvent the shrinking of the spatial dimensions when performing convolution. The amount of zeros added on each side for every spatial dimension is an additional hyperparameter P. An example of zero-padding is illustrated in Figure 2.13.



(a) Stride $S = 1$.



(b) Stride $S = 2$.



(c) Stride $S = 3$.

Figure 2.12: Convolution on a one-dimensional input with different strides.

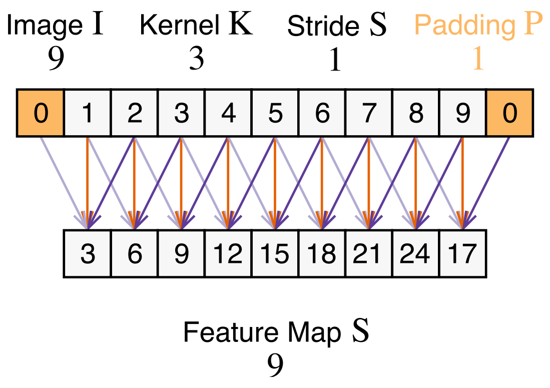
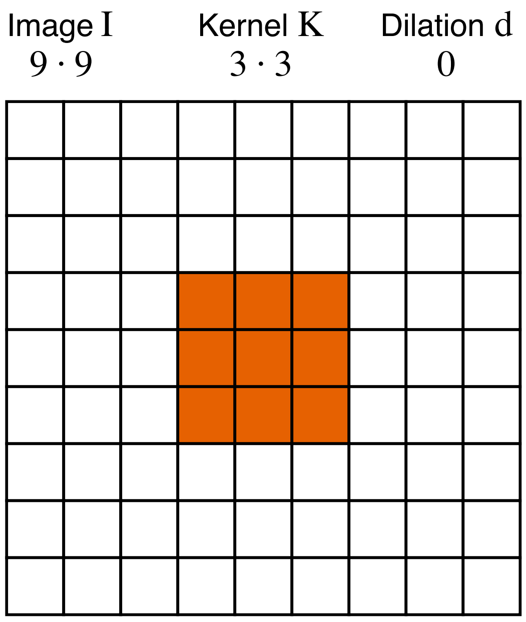
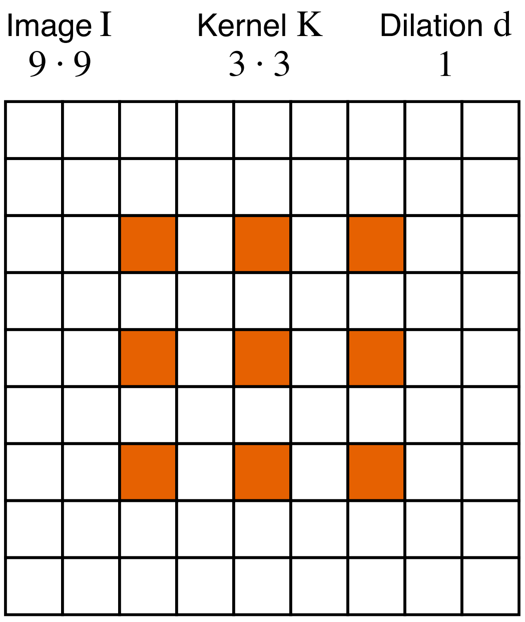


Figure 2.13: One-dimensional convolution with zero-padding.

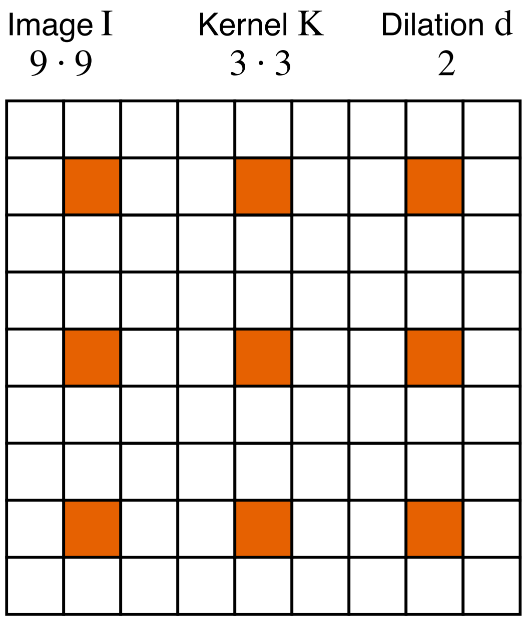
Dilation Introduced very recently, the *dilation* d is another hyperparameter that allows for the convolutional layer to have a larger effective receptive field relative to the input while keeping the kernel size the same. This is achieved by introducing d spaces between each cell of the kernel ([9], [75]). Standard convolution, as described previously, simply uses a dilation of 0, and therefore uses a contiguous kernel. By increasing the dilation, and therefore the spacing between kernel cells, it is possible for a convolutional layer to pick up a larger spatial extent of the input while keeping memory consumption the same. The concept of *dilated convolutions* (sometimes also called *atrous convolutions*) with different dilations is illustrated in Figure 2.14.



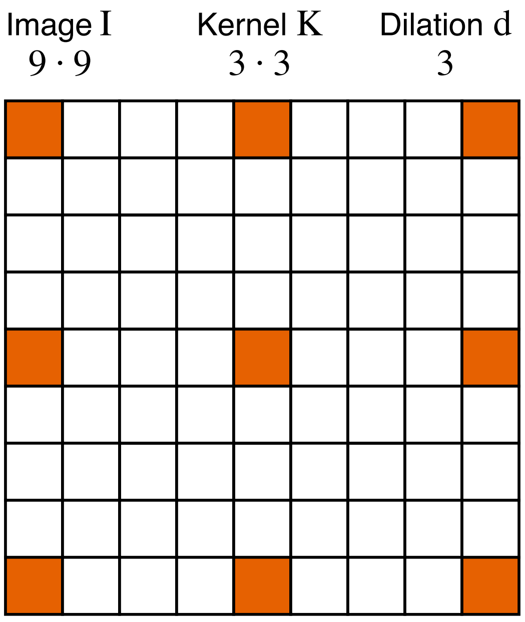
(a) Dilation $d = 0$.



(b) Dilation $d = 1$.



(c) Dilation $d = 2$.



(d) Dilation $d = 3$.

Figure 2.14: Dilated convolution on a two-dimensional input for different dilations. For illustrative purposes, only one kernel position, centered on the input image is visualized.

Given the input volume size W, the kernel size K, the stride S, dilation d and the zero-padding P, the resulting output volume has the size of[2]

$$W_o = \left\lfloor \frac{W + 2P - K - (K-1)(d-1)}{S} \right\rfloor + 1. \tag{2.39}$$

By carefully utilizing the kernel size, number of filters, stride, dilation and zero-padding, it is possible to fully define the spatial and feature dimensions of the output while still being able to adjust the amount of weights and effective receptive field per convolutional layer, giving fine-grained control over every part of the convolutional layer.

2.5.2 Pooling

In order to keep the amount of parameters low and to further increase the effective receptive field of outputs with respect to the input, it can be beneficial to use a special form of spatial downsampling, called *pooling*, after some convolutional layers in the network [40].

Pooling, similar to convolution, can also intuitively be understood as a sliding kernel mechanism, with similar parameters, such as stride and kernel size. The key difference is that pooling computes a fixed function on its inputs, which is most commonly the max operation (*max-pooling*). The most common form of pooling consists of a $[2 \times 2]$ kernel with a stride of 2. When sliding this kernel across the input volume using the max function, it effectively processes non-overlapping $[2 \times 2]$ chunks of the input volume and only keeps the largest value at the output feature map, discarding 75% of the input data. Another common variant of pooling is *average pooling*, for which the kernel simply computes the average over all its input elements. The backpropagation for max-pooling can simply be done by only routing the gradient through the input that had the largest value in the forward pass [40]. An example of max-pooling can be seen in Figure 2.15.

As this function is fixed, it does not require any trainable parameters, and therefore does not increase the memory consumption and model capacity of the *CNN* architecture, compared to strided convolutions. However, recent *CNNs* architectures seem to steer away from using pooling for downsampling, and instead propose to always use strided convolutions to reduce the spatial dimensions [66]. This seems to be especially important when training generative models such as Generative Adversarial Networks (GANs) [52].

2.5.3 Fractionally Strided Convolution

Sometimes, similarly to how it can be useful to reduce the spacial resolution of intermediate layers in a *CNN*, it can also be useful to increase spatial resolution, for example in encoder-decoder architectures ([54], [5]). While it is also possible to simply increase the spatial resolution by standard image processing methods such as nearest neighbor or bilinear

[2]Convolution Arithmetic Tutorial, `http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html`, Accessed: 01.02.2018

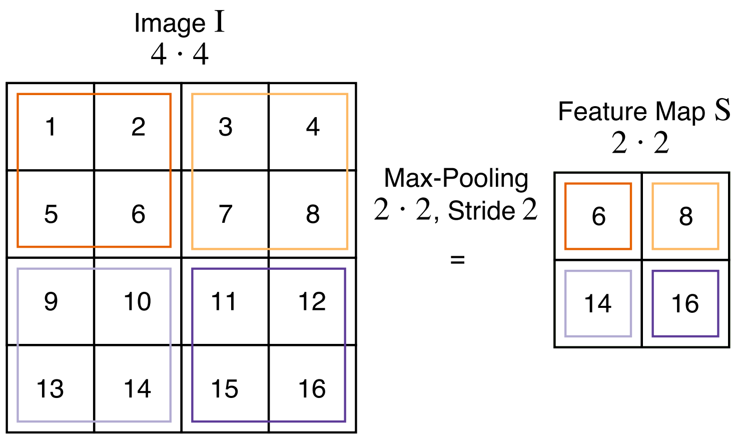


Figure 2.15: Two-dimensional max-pooling, using a pooling size of [2 × 2] with a stride of 2.

upsampling, it can be useful to combine the upsampling process with learnable parameters, intuitively 'learning' the upsampling algorithm which best fits the given data.

Fractionally Strided Convolutions, also sometimes called *Transposed Convolutions*, simply swap the forward and the backward pass of a regular convolution, therefore leading to a larger spatial resolution given specific parameters.

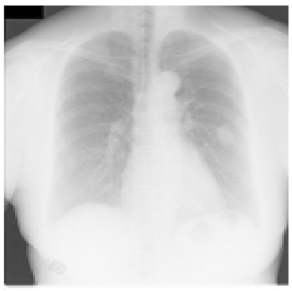
2.6 Image Segmentation using Deep Convolutional Neural Networks

Due to the strong focus of this thesis on image segmentation, this section will give a brief introduction to the problem of image segmentation and how to approach it from a deep learning perspective using encoder-decoder *CNN* architectures.

2.6.1 Image Segmentation

The goal of image segmentation is to partition an image into a set of disjoint regions that cover it [68]. Typically, the output of image segmentation is a segmentation mask, where every pixel contains the label of the region or class detected by the segmentation algorithm. Therefore, image segmentation is a pixel-wise operation. For every input pixel, the goal is to produce an output pixel which has the correct region label. For example, if the goal is to segment a lung X-ray image into the regions 'left lung' and 'everything except left lung', the result should ideally look like illustrated in Figure 2.16.

The easiest method to perform binary (two-class) image segmentation is by simply *thresholding* the image. If the input pixel is larger than a specific threshold, it belongs to the foreground class, otherwise it belongs to the background class. However, this only works for very easy tasks, where the regions of interest share a common pixel intensity


(a) Lung X-ray image.

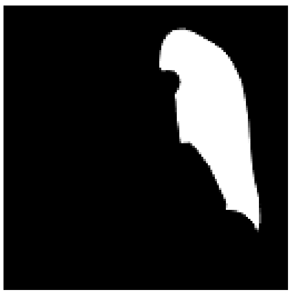

(b) Groundtruth segmentation mask for the left lung.

Figure 2.16: Image segmentation of the left lung of a lung X-ray image.

range. Image segmentation is one of the oldest and most relevant computer vision problems, as it reduces complexity, and is therefore a valuable preprocessing step for many tasks. As an example, the segmentation of the left lung significantly reduces the search area if the task is to find anomalies in the left lung. Before the rise of deep learning, a long history of research led to image segmentation algorithms based on thresholding, clustering, region growing, graph partitioning and many more [68]. However, since modern deep learning approaches achieve state-of-the-art on image segmentation tasks, most recent advancements in image segmentation have been in the deep learning domain [15].

2.6.2 Encoder-Decoder Architecture

As previously described, image segmentation is a pixel-wise method, therefore it is necessary for a deep neural network to take an input image, and produce an output segmentation mask. Similar to most modern deep neural networks, *CNNs* are typically chosen for this task, as the sheer amount of pixels would make fully-connected *FNNs* require too much memory in most cases.

A very common network architecture for image segmentation is the *encoder-decoder* network. As the name implies, such a network mainly consists of two stages, the *encoder*, and the *decoder*. The encoder typically consists of a series of convolutions and pooling operations, and the goal of this stage is to increase the effective receptive field of the network as well as to generate an embedding in a lower dimensional feature space. For every pooling step, the following convolutional layers will have a larger receptive field relative to the input image, which is important for the network to learn global image context for regions that are large. The decoder is typically a mirrored version of the encoder, and contains a series of upsampling operations (or fractionally strided convolutions) and convolutions. The input to the decoder is the output of the encoder, which is the layer representing the lower dimensional, compressed feature embedding. The goal of the decoder is to 'reconstruct' a segmentation mask from the highly compressed output of the encoder. This is mainly motivated by the idea that by compressing the input to a small, intermediate representation, the network has to learn a good representation of the input data, otherwise the decoder will not be able to construct a good segmentation mask out of the compressed intermediate representation. A very popular convolutional architecture of this kind, especially in the medical computer vision community, is the U-Net architecture [54]. In addition to containing standard encoder and decoder paths, the U-Net also contains 'shortcuts', which are typically implemented as channel concatenations or additions, between encoder and decoder at the same depth. This allows for the decoder to incorporate finer details of the encoder stage as well, and not just strictly use upsampled versions of the final encoder output. A U-Net style network architecture is illustrated in Figure 2.17.

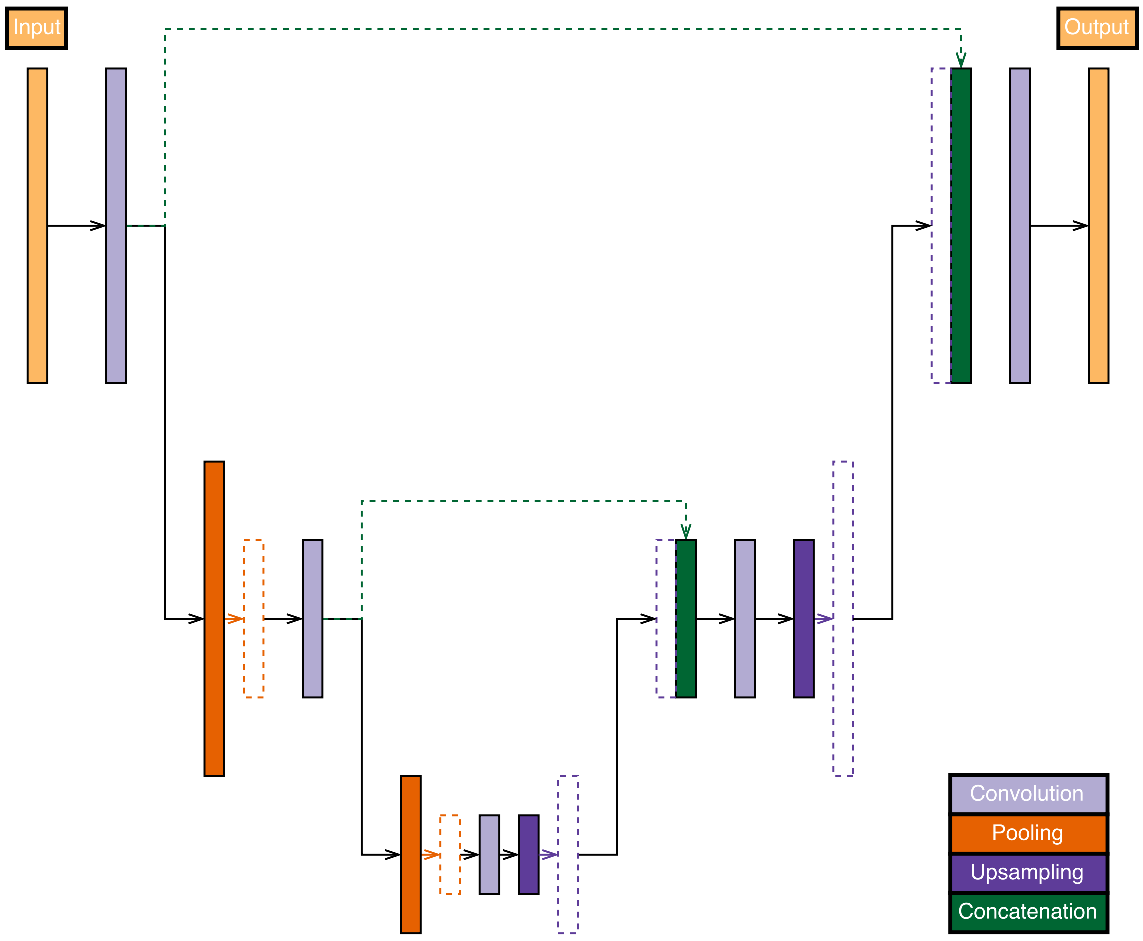


Figure 2.17: U-Net style encoder-decoder network architecture.

2.6.3 Softmax Cross-Entropy Loss

As previously discussed, optimizing neural networks via gradient descent requires a loss function, for which the gradients are computed. For image segmentation, a common choice is to use the *softmax cross-entropy loss*, which is also commonly used for image classification. The only difference between image segmentation and image classification regarding the loss function is that for image segmentation, the loss is computed per pixel individually, while for image classification, the loss is computed for the whole image. Instead of a final segmentation mask, segmentation networks typically predict 'pseudo-probabilities' for every pixel and class. Therefore, the network output typically consists of a number of images stacked on the depth axis, one image for each possible class, where each image predicts a 'pseudo-probability' of every pixel in the image belonging to the respective class. Equivalently, every output pixel has a vector of 'pseudo-probabilities' containing the likelihood of it belonging to a certain class.

The softmax cross-entropy loss consists of two parts. First, the network outputs (the 'pseudo-probabilities') are fed through a *softmax* function, which is simply a multidimensional generalization of the sigmoid function. This results in actual probabilities for every pixel, i.e. the sum over all class probabilities for every pixel is 1. Given a vector $\boldsymbol{z_j}$ of 'pseudo-probabilities' containing K class entries for a single pixel j of the output minibatch, the softmax function is computed as [47]

$$\sigma(\boldsymbol{z_j})_i = \frac{e^{z_{j_i}}}{\sum_k^K e^{z_{j_k}}}, \tag{2.40}$$

where i is the class index.

The second part of the softmax cross-entropy loss is the computation of the *cross-entropy* between the now computed softmax outputs and the target probabilities. In image segmentation, pixels can typically only belong to a single class, therefore the target probabilities are simply 1.0 for the class a pixel belongs to, and 0.0 otherwise. Given a target vector of probabilities $\boldsymbol{t_j}$ and the softmax output $\boldsymbol{\sigma(z_j)}$, the cross-entropy for a single pixel j of the output minibatch can be computed as [47]

$$C_j = -\sum_i^K t_{j_i} \log(\sigma(\boldsymbol{z_j})_i), \tag{2.41}$$

where i is the class index. Finally, by computing the cross-entropy C_j for every target pixel j in the whole minibatch, the full cross-entropy loss is computed by averaging all cross-entropies for every pixel:

$$C = \frac{1}{N}\sum_j^N C_j, \tag{2.42}$$

where N is the total number of pixels of the output minibatch, and j is the pixel index over the whole minibatch.

2.7 Data Augmentation

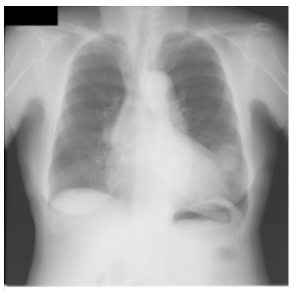
For some applications, acquiring a sufficient amount of training data for training a deep neural network can be difficult. As an example, in medical imaging, the acquisition of labeled training data is very time-consuming and costly, since a trained expert needs to manually annotate every image in the training set. An insufficient amount of training data can lead to overfitting, as the neural network is more likely to just memorize certain aspects of the training set. To regularize this type of overfitting, *data augmentation* is commonly applied. Data augmentation describes the process of generating additional training data by transforming the given input training data. Most commonly, data augmentation for deep learning is done *online*, therefore the input image (or input image minibatch for *SGD*) is transformed directly before being fed into the deep neural network. Data augmentation has been shown to be especially beneficial for medical image segmentation [54]. Even for large datasets such as ImageNet [59], it has been shown that data augmentation can be beneficial as an additional regularizer for very deep architectures [34]. Additionally, data augmentation allows for an easy way to incorporate prior knowledge about possible unseen data. For example, if test images are taken with a varying amount of brightness, it might make sense to augment with random intensity shifts to accomodate for the variation in brightness in test data. Possible data augmentation schemes range from simple additive or multiplicative image modifications such as intensity shifts, to geometric transformations such as rotation, scaling, elastic deformation to synthetic data generation. This section discusses commonly used data augmentation methods and their possible uses and effects on training deep neural networks. For all data augmentation schemes in this section, the parameters assume an image minibatch $\boldsymbol{I}$ normalized to an intensity range of $[-1, 1]$ with a spatial extent of $[256 \times 256]$. All described data augmentation schemes should only be carefully applied in the context of task, dataset and network architecture, as augmentation parameters that do not fit the training setup can lead to worse performance than not using augmentation at all.

2.7.1 Additive Noise

Additive noise describes the augmentation scheme of adding element-wise noise from a specific distribution to an image minibatch before feeding it into the neural network. As an example, given a minibatch of noise-matrices $\boldsymbol{N} \sim \mathcal{N}(\mu, \sigma^2)$ drawn from a zero-mean Gaussian noise distribution with a standard deviation of $\sigma = 0.2$, this results in the following augmented image minibatch:

$$\boldsymbol{I_A} = \boldsymbol{I} + \boldsymbol{N} \tag{2.43}$$

An example of an augmented image using additive noise can be seen in Figure 2.18.


(a) Base Image.

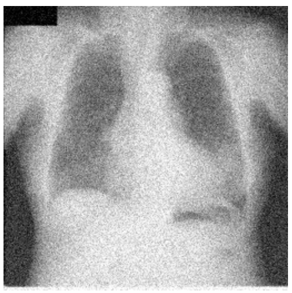
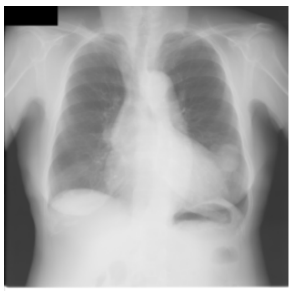

(b) Image augmented with additive zero-mean Gaussian noise with standard deviation of $\sigma = 0.2$.

Figure 2.18: Additive noise for data augmentation.

Adding noise to the input forces the neural network to be tolerant against uncorrelated variations in pixel values, and therefore can reduce overfitting. Additionally, if the noise source of test images is known, additive noise can help to adjust training images to be more similar to test images.

2.7.2 Intensity Scaling

For *intensity scaling*, each image is transformed by element-wise multiplication with a scalar. This typically results in a change of image *contrast*, as multiplying the image with a scalar has a more significant impact (when looking at the absolute change in intensity) on larger image intensities. An example for intensity scaling by a scalar drawn from a Gaussian distribution with a mean of $\mu = 1$ and a standard deviation of $\sigma = 0.4$ is shown in Figure 2.19.


(a) Base Image.

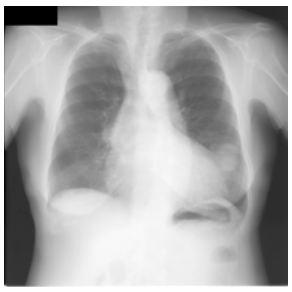
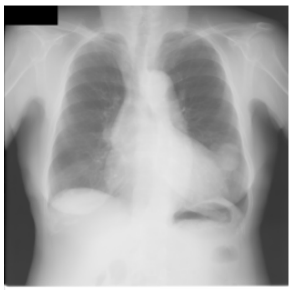

(b) Image augmented with intensity scaling using Gaussian noise with a mean of $\mu = 1.0$ and standard deviation of $\sigma = 0.4$.

Figure 2.19: Intensity scaling for data augmentation.

Intensity scaling can be useful if test images are expected to have varying contrast, as the network can be made more tolerant against changes in contrast if this augmentation method is applied.

2.7.3 Intensity Shift

Intensity shift describes the procedure of performing an element-wise addition of a scalar to the image. This corresponds to a basic *brightness* change for the image, as positive values for the intensity shift lead to brighter images, while negative values lead to darker images. An example for intensity shift by a scalar drawn from a Gaussian distribution with zero-mean and a standard deviation of $\sigma = 0.4$ is shown in Figure 2.20.


(a) Base Image.

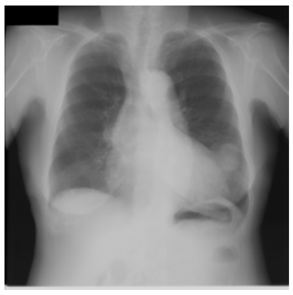

(b) Image augmented with intensity shift using zero-mean Gaussian noise with standard deviation of $\sigma = 0.4$.

Figure 2.20: Intensity shift for data augmentation.

This augmentation technique is especially useful to improve generalization of neural networks for varying brightness in the test data.

2.7.4 Random Flipping

For *random flipping*, images are typically flipped around a chosen axis, most commonly at a rate of 50%. The resulting minibatch therefore has an equal chance of containing the flipped or non-flipped version of the image. An example for a horizontally flipped image is shown in Figure 2.21.

Random flipping is very useful when the test data contains, for example, natural images or faces, as these images typically are still valid when flipped.

2.7.5 Random Translation

When performing *random translation*, the image is shifted by a random distance on each axis. For this operation, there are several ways to decide how the pixels outside the image boundaries (which can now be visible due to the translation) are computed. The *nearest neighbor* mode simply repeats the nearest pixel which is inside the original image

(a) Base Image. (b) Image flipped horizontally.

Figure 2.21: Horizontal flipping for data augmentation.

boundaries. The *constant* mode just sets all outside pixels to a certain constant value. Using the *wrap* mode, the outside pixels are computed by 'wrapping' the image around each boundary. Similarly, by using the *reflect* mode, the outside pixels are computed by reflecting the image around each boundary. An example for random translation with the reflect boundary mode is shown in Figure 2.22.

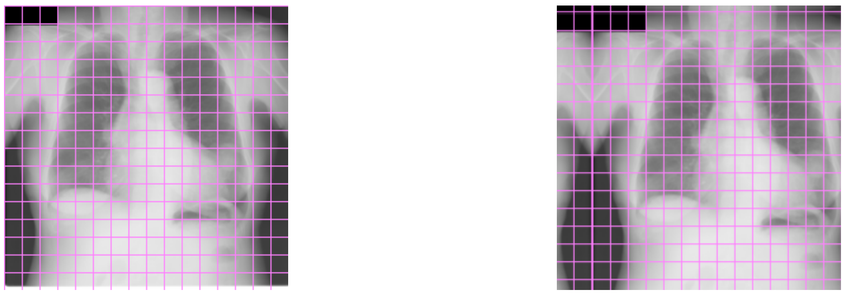
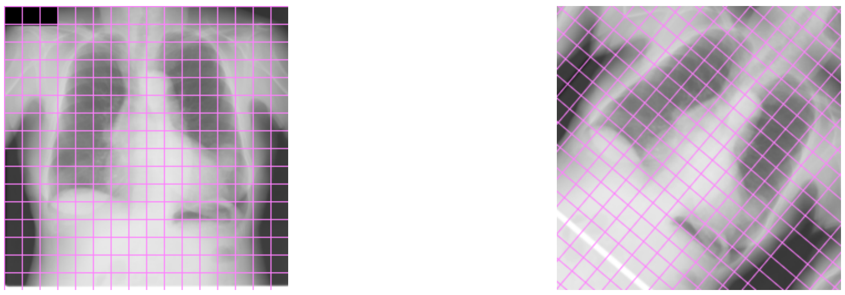

(a) Base Image. (b) Image translated randomly using a shift drawn from a Gaussian distribution with a mean of $\mu = 15$ pixels and standard deviation of $\sigma = 15$ pixels for each axis.

Figure 2.22: Random translation for data augmentation. The magenta grid is just for illustrative purposes, to better visualize how the image coordinates were transformed.

While *CNNs* in itself are translation invariant due to the convolution operation, it can still be useful to augment data using random translation, as this can force the learned kernels to generalize better to different image positions, when combined with other geometric intensity transformations.

2.7.6 Random Rotation

Similar to random translation, *random rotation* is a geometric coordinate transformation applied on the input image minibatch. In random rotation, each image inside the minibatch is rotated randomly, most commonly either using fixed possible angles (e.g. 0, 90, 180, or 270 degrees) or using a randomly drawn angle in a specific range (e.g. a range between -60 and 60 degrees). As rotation can also lead to pixels outside the image now being inside the image, the same boundary modes as mentioned in Section 2.7.5 apply. An example of an image augmented with random rotation in the range of $(-60, 60)$ degrees using the reflect mode can be seen in Figure 2.23. Random rotation is very useful for making the neural network invariant to rotation.



(a) Base Image.

(b) Image rotated randomly using a rotation angle drawn from a uniform distribution in the range of $(-60, 60)$ degrees.

Figure 2.23: Random rotation for data augmentation. The magenta grid is just for illustrative purposes, to better visualize how the image coordinates were transformed.

2.7.7 Random Cropping

In *random cropping*, each image inside a minibatch is cropped to a smaller size. This has multiple effects: First, the image size is reduced, therefore requiring a lower amount of memory for storing neural network activations. Therefore, cropping can be a valid alternative to simple downsampling, with the benefit of not losing fine-grained information. The second effect is that it forces the neural network to not expect certain features at certain positions all the time, as the crop might not even contain specific regions. An example of a randomly cropped image (to a size of $[128 \times 128]$) can be seen in Figure 2.24.

In addition to the regularizing effect, random cropping also does not hinder the testing on new images. For the results on large, uncropped test images, the image can simply be cropped multiple times in a way that the whole image is covered by crops. The final results are then obtained by assembling the crop-wise results. As re-assembling of cropped results can lead to inconsistencies at the edges (e.g. for pixel-wise methods such as segmentation), in practice, the test image can be cropped such that there is an overlap between crops. For classification, the final label can be decided by majority vote, while for pixel-wise methods, a fully sized image can be assembled by computing cropped partial images, using interpolation in the overlapping regions of every crop.

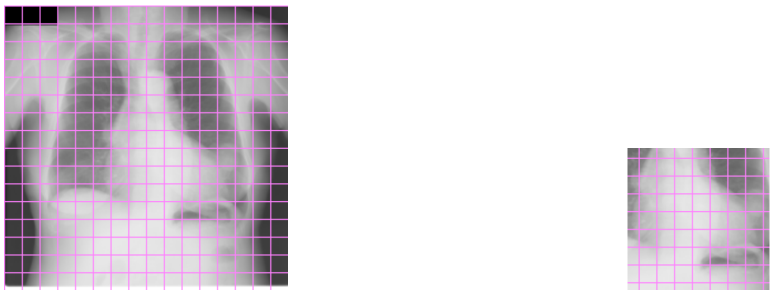

(a) Base Image. **(b)** Image cropped randomly to a size of [128 × 128].

Figure 2.24: Random cropping for data augmentation. The magenta grid is just for illustrative purposes, to better visualize how the image was cropped.

2.7.8 Elastic Deformation

In addition to the simpler geometric transformations previously described, *elastic deformation* is a method to deform images using basis spline transformations. This method is particularly common in medical imaging, as a tuned spline transformation can better approximate anatomical variation compared to very simple transformations [54]. The method works by defining a grid of sparse, regular control points and deforming each control point using basis spline interpolation kernels[3]. An example of elastic deformation is shown in Figure 2.25.

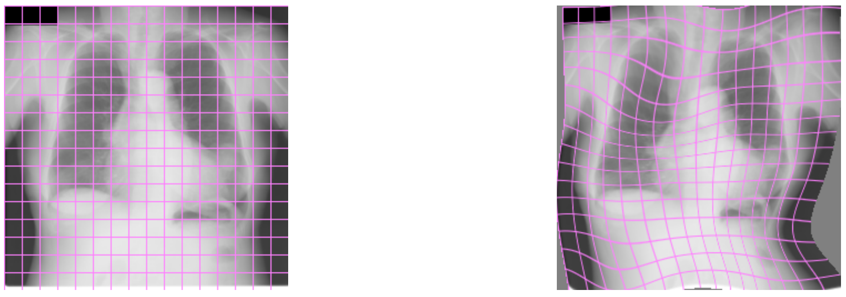

(a) Base Image. **(b)** Image augmented using elastic deformation with displacements of the basis spline transformation drawn from a zero-mean Gaussian distribution with a standard deviation of $\sigma = 15$ pixels.

Figure 2.25: Elastic deformation for data augmentation. The magenta grid is just for illustrative purposes, to better visualize how the image coordinates were transformed.

While elastic deformation can be very useful to model realistic variation in the data, it is very computationally intensive, and therefore should only be used when appropriate,

[3]ITK: Insight Segmentation and Registration Toolkit - Documentation, `https://itk.org/Doxygen/html/classitk_1_1BSplineTransform.html`, Accessed: 01.02.2018

as it significantly increases the training time of the neural network in the online data augmentation case.

2.7.9 Synthetic Data Augmentation

Synthetic data augmentation describes a completely different approach to standard data augmentation. In this method, the input images are not transformed, but instead a generative model of some kind is tuned to generate additional, synthetic data resembling the real data. Very recently, 3D models have been used to render highly realistic images to increase the amount of training data and improve the performance of object detection [56]. As another example, rendered synthetic images have been used to improve the performance of landmark detection in medical imaging [53]. Furthermore, it has been shown that using modern deep learning models such as *GANs*, it is even possible to refine purely rendered data with information from real data to arrive at more realistic synthetic data [64]. As this method is especially relevant to our investigation, Section 3.4 describes this approach in more detail. The main drawback of synthetic data augmentation using rendered images or models is that it requires a significant time investment. It is not trivial to create a renderer for images that is realistic enough to be useful for training neural networks, and the refinement to make the rendered images more realistic requires a lot of fine-tuning and careful parameterization to arrive at good results.

2.8 Summary

It is undeniable that *deep learning* resulted in significant advancements in the computer vision community. Whether it is object classification, image segmentation, landmark localization or image translation, most computer vision applications have benefitted significantly from deep learning. In this chapter, we described the basic building blocks of deep learning, the *artificial neurons*, as well as their computational behavior. Additionally, we discussed how to organize multiple artificial neurons into *FNNs*, and how to efficiently optimize *FNNs* using *SGD*, backpropagation and various other optimization and regularization schemes. Furthermore, we described *CNNs*, and the main differences between regular *FNNs* and *CNNs*, such as *convolution*, *pooling* and *fractionally strided convolutions*. As we have a strong focus on image segmentation in this thesis, we illustrated how modern *CNNs* can be used for image segmentation. Finally, as the main focus of this thesis, *data augmentation* was discussed in detail, and some of the most common data augmentation methods were described.

3 Generative Adversarial Networks

Contents

3.1 Fundamentals **44**
3.2 Deep Convolutional Generative Adversarial Networks **47**
3.3 Wasserstein Generative Adversarial Networks **49**
3.4 SimGAN **53**
3.5 Summary **56**

Since the introduction of Generative Adversarial Networks (GANs) in 2014 by Goodfellow et al. [19], the original idea has been improved and iterated upon, and it has been shown that *GANs* are useful in a multitude of applications, such as domain-transfer ([30], [78]), synthetic data generation and refinement ([64], [45]), super-resolution [39] or state-of-the-art high-resolution image generation ([21], [31]). While many of these applications iteratively improved the common knowledge about the inner workings of *GANs*, this chapter will only focus on the major milestones in *GAN* research relevant to our work. Section 3.1 describes the original approach, as introduced by Goodfellow et al. in 2014 [19]. In Section 3.2, we discuss the advancements made by Radford et al. in 2015 culminating in the Deep Convolutional Generative Adversarial Network (DCGAN) [52], which, for the first time, provided a set of guidelines for efficient and stable training of *GANs*. Since the theoretical background of *GANs* was still relatively unexplored, researchers started to take a closer look at the theoretical formulation of *GANs* ([48],[77]). This lead to the introduction of multiple variants of different optimization criterions for training *GANs*, including least squares loss [42], matching of reconstruction loss distributions [7] or the Wasserstein distance as a loss function ([3],[21]). As our experiments described in Chapter 4 are based on the Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) introduced by Gulrajani et al. in 2017 [21], we describe Wasserstein Generative Adversarial Networks (WGANs) [3] and *WGAN-GP* [21] in more detail in Section 3.3. Finally, as an example of *GANs* applied in the context of data augmentation, we describe *SimGAN* [64] in more detail in Section 3.4.

3.1 Fundamentals

GANs, as introduced by Goodfellow et al., currently represent the state-of-the-art for generative models. A *GAN* is a framework for estimating generative models, consisting of two differentiable submodels, which are typically implemented as deep neural networks: the *generator* G with parameters $\boldsymbol{\theta_G}$ and the *discriminator* D with parameters $\boldsymbol{\theta_D}$. An illustration of a typical *GAN* setup is shown in Figure 3.1. The generator and the discriminator compete against each other: the generator is trained to generate images $G(\boldsymbol{z})$ which resemble the training data distribution p_R, using a latent noise vector $\boldsymbol{z}$, sampled from the distribution p_z as input, while the discriminator receives the generated images $G(\boldsymbol{z})$ as well as the real training data $\boldsymbol{x}$ as input and is trained to differentiate between generated images and real images. Therefore, G is trained to minimize the probability of D identifying the generated images as synthetic $(D(G(\boldsymbol{z})) \approx 0)$, while D is trained to maximize the probability of itself being correct $(D(G(\boldsymbol{z})) \approx 0)$, $(D(\boldsymbol{x}) \approx 1)$, leading to the following minimax [20] value function:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_R}\left[log D(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim p_z}\left[log\left(1 - D\left(G\left(\boldsymbol{z}\right)\right)\right)\right] \tag{3.1}$$

The ideal discriminator for a fixed generator, as shown by Goodfellow et al., is given by

$$D_G^* = \frac{p_R}{p_R + p_G}, \tag{3.2}$$

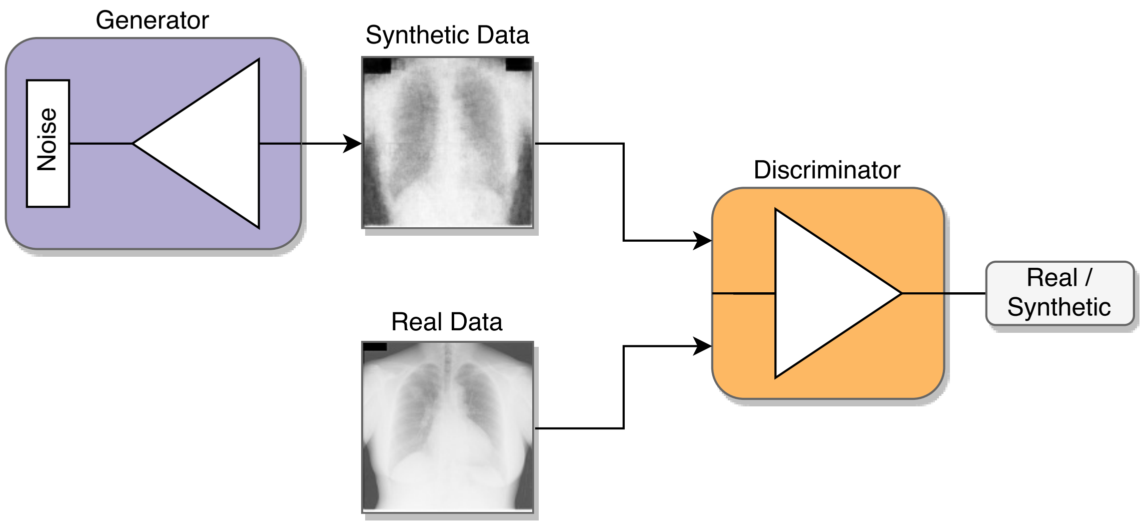


Figure 3.1: Architecture setup for a typical *GAN*.

where p_G is the distribution of data generated by the generator. The optimum of the minimax game is reached for $p_R = p_G$, i.e. when the generator perfectly replicates the training data distribution. For this optimum, the optimal discriminator output is $D^*_G = \frac{1}{2}$, meaning that the discriminator is unable to differentiate between training data and generated data distributions. For practical implementation of *GANs*, the minimax objective shown in Equation 3.1 is optimized by alternating the optimization of G and D, optimizing G once for multiple steps of D, in order to keep D in its optimal region, leading to Algorithm 2:

Algorithm 2: Training algorithm for *GANs*, using minibatch stochastic gradient descent. For each generator training step, k discriminator training steps are performed.

```
for number of training iterations do
    for k discriminator update steps do
        Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
        Take a minibatch $\boldsymbol{x}$ of $m$ images from the real data;
        Compute the weight update for the discriminator $D$ by ascending the
          stochastic gradient $\boldsymbol{g_{\theta_D}}$:
            $$\boldsymbol{g_{\theta_D}} = \nabla_{\boldsymbol{\theta_D}} \frac{1}{m} \sum_{i=1}^{m} \left[ logD\left(x^{(i)}\right) + log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right] \quad (3.3)$$
            $$\boldsymbol{\theta_D} = \boldsymbol{\theta_D} + \eta\, \boldsymbol{g_{\theta_D}} \quad (3.4)$$
    Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
    Compute the weight update for the generator $G$ by descending the stochastic
      gradient $\boldsymbol{g_{\theta_G}}$:
        $$\boldsymbol{g_{\theta_G}} = \nabla_{\boldsymbol{\theta_G}} \frac{1}{m} \sum_{i=1}^{m} log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \quad (3.5)$$
        $$\boldsymbol{\theta_G} = \boldsymbol{\theta_G} + \eta\, \boldsymbol{g_{\theta_G}} \quad (3.6)$$
```

A significant part of the popularity and wide adoption of *GANs* in research is that *GANs* can be trained by using well researched and understood methods also used for training discriminative models, such as backpropagation and Stochastic Gradient Descent (SGD), without requiring intractable probabilistic computations [19]. The first examples of *GANs* used popular deep learning techniques such as Dropout [67] and Rectified Linear Units (ReLUs) [17] in their architecture, which was only easily possible due to the generator and discriminator being neural networks. Examples of images generated by these early networks can be seen in Figure 3.2 and Figure 3.3.

While the results seen in Figure 3.2 and Figure 3.3 were competitive with other generative models at the time, the stability of training *GANs* was a significant issue. The

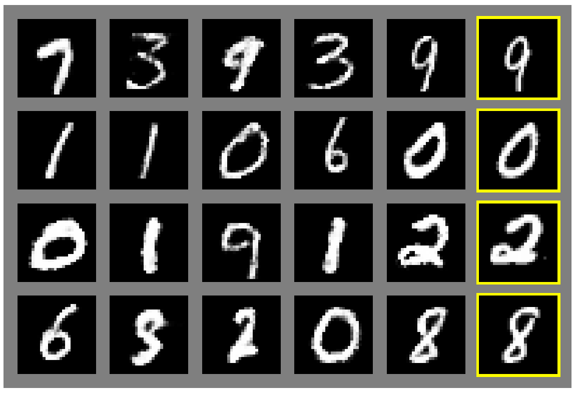


Figure 3.2: Samples from a fully-connected Generative Adversarial Network trained on MNIST [38]. Image taken from [19].

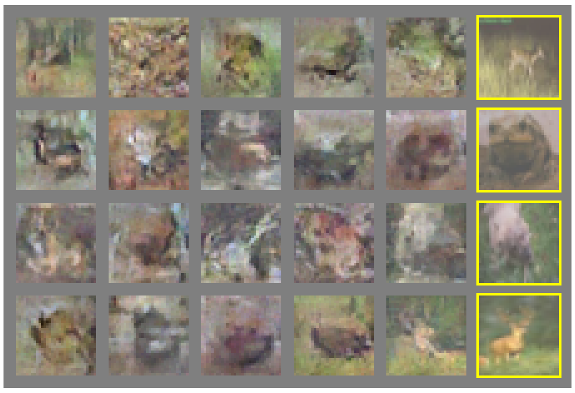


Figure 3.3: Samples from a Generative Adversarial Network with a convolutional generator and discriminator trained on CIFAR10 [33]. Image taken from [19].

optimization between the generator and the discriminator was very fragile, as one of the networks could sometimes overpower the other, leading to a multitude of possible failure states of *GANs*, for example mode collapse [20]. Mode collapse describes a common problem for *GANs*, where the generator learns to map multiple different noise input vectors $\boldsymbol{z}$ to the same output $G(\boldsymbol{z})$. This is mainly because the gradients in the discriminator are computed independently from each other, without incorporating any kind of similarity measure for comparing samples of a given minibatch [60]. Therefore, the generator might learn to only create a single image (mode) with the largest discriminator response. When this situation settles in, the discriminator quickly learns that this single mode comes from the generator, and as a response, the generator will shift this mode to a different location. However, since all of the generated images in a mode-collapsed minibatch are almost identical, the difference in gradients is not enough to force the generator to create multiple different images, it will just generate a different single image instead, leading to an oscillatory failure state. Partial mode collapse similarly results in the generator creating only a very small amount of different images, or images which all share common features. An example of a generator exhibiting mode collapse can be seen in Figure 3.4. Without extensive hyperparameter searches, this restricts the use of early *GAN* implementations to applications where the variance of generated output images is not an important criterion. Fortunately, this was adressed in future *GAN* research, by using methods such as minibatch discrimination [60] or different optimization objectives such as the Wasserstein loss ([3],[21]) which is described in more detail in Section 3.3.

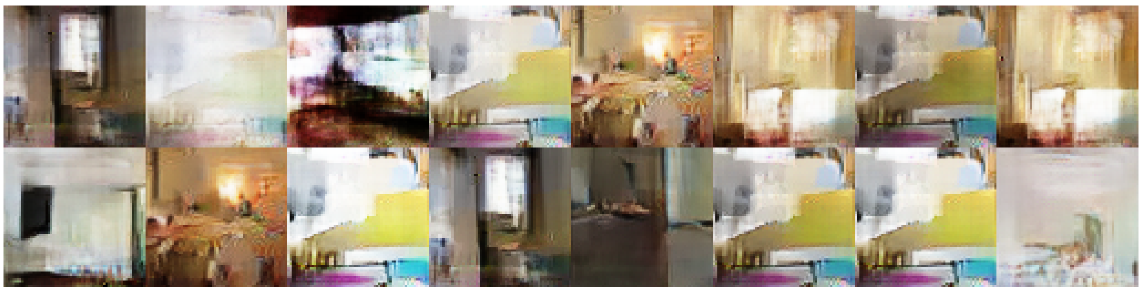


Figure 3.4: Samples from a Generative Adversarial Network exhibiting mode collapse trained on LSUN [76] bedrooms. Multiple similar images are generated in the same minibatch instead of a diverse set of samples. Image taken from [3].

3.2 Deep Convolutional Generative Adversarial Networks

Building on the success of *GANs*, Radford et al. introduced Deep Convolutional Generative Adversarial Network (DCGAN) [52], with the goal to better incorporate recent methods used in training Convolutional Neural Networks (CNNs). The *DCGAN* architecture is built on a set of guidelines to improve stability of training and image quality:

Fully-Convolutional Neural Network (FCNN) architecture Use strided convolutions instead of pooling and fractionally strided convolutions instead of upsampling to make the generator and discriminator architectures fully convolutional [52]. An example of such an architecture in form of the *DCGAN* generator is demonstrated in Figure 3.5. The discriminator architecture is simply mirrored.

Batch Normalization [29] Add Batch Normalization to all convolutional layers except the final generator layer and the first discriminator layer, as it is necessary for deep networks to begin learning and improve training stability.

Activation Functions Use *ReLU* [17] activation in all generator layers except the last one, which uses Hyperbolic Tangent (tanh) activation. Use Leaky Rectified Linear Unit (leaky ReLU) [41] activation in all discriminator layers.

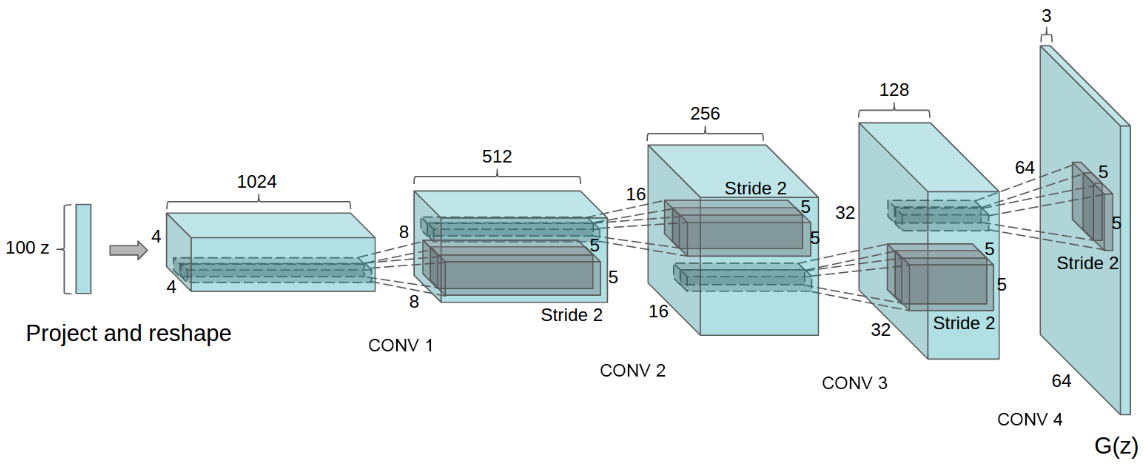


Figure 3.5: *DCGAN* fully convolutional generator architecture. Image taken from [52].

To demonstrate their guidelines for designing *GAN*, Radford et al. trained a *DCGAN* on the LSUN [76] bedrooms dataset, which contains over 3 million training examples at an image resolution of $[64 \times 64]$. Fully converged samples from the experiment performed by Radford et al. can be seen in Figure 3.6. Furthermore, Radford et al. also experimented with the latent noise vector input of the generator. Drawing two random noise vectors $\boldsymbol{z_1}$ and $\boldsymbol{z_2}$ from a uniform distribution and linearly interpolating between them, they showed that the generated images between $\boldsymbol{z_1}$ and $\boldsymbol{z_2}$ form a smooth transition without sharp edges, suggesting that the generator did not just learn to recreate the training images, but has succeeded in establishing a representation of the submanifold spanned by the input images. In addition to this experiment, it was also shown that the properties of the generator latent space allow for vector arithmetic for visual concepts. For example, when computing the average noise vector input for output images where people are wearing glasses, this average can be used to add glasses to a person without glasses, by simply adding the average noise vector of people with glasses to the vector of a person without glasses.

An additional key motivation for *DCGAN* was to use *CNNs* in the field of unsupervised learning. Radford et al. showed that by using the discriminator as a feature extractor for image classification tasks, they achieve competitive performance with other unsupervised algorithms on CIFAR10 [33] and SVHN [46].

While the training of *GANs* is more stable when using *DCGAN* architectures, the hyperparameters for training *DCGAN* (model size, architecture, optimizer parameters, ...) still have to be chosen very carefully. Otherwise, the *DCGAN* might oscillate during training, where it simply switches between generating noise artefacts and generating reasonable images, which was also reported by the authors of *DCGAN*.

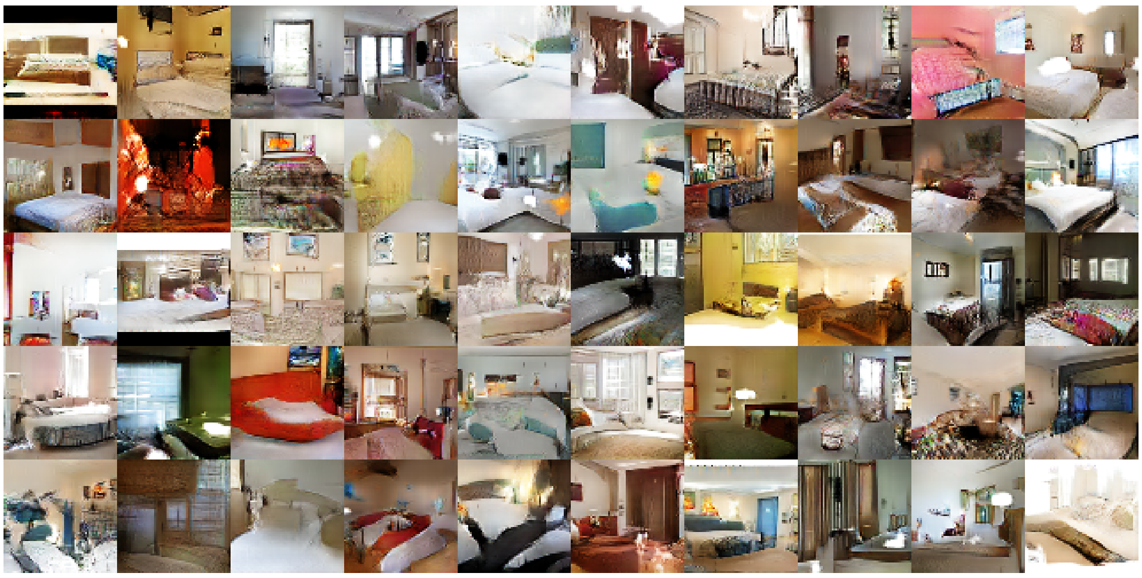


Figure 3.6: *DCGAN* samples after five epochs of training on LSUN [76] bedrooms. Image taken from [52].

3.3 Wasserstein Generative Adversarial Networks

With the advent of theoretical *GAN* research ([48],[77]), new loss functions for optimizing *GANs* became more popular. This led to the introduction of Wasserstein Generative Adversarial Networks (WGANs) [3] by Arjovsky et al. in 2017. Often, learning a probability distribution is done by using maximum likelihood estimation to find the best fit of the model density $P_{\boldsymbol{\theta}}$ to the given real data $\boldsymbol{x}$. Given the real data distribution p_R and the model distribution p_G, the limit (in the number of samples) of the maximum likelihood estimate is equal to minimizing the Kullback-Leibler (KL) divergence $KL(p_R||p_G)$. However, the *KL* divergence is undefined or infinite if the model density $P_{\boldsymbol{\theta}}$ does not exist, for example if the distributions are supported by low dimensional manifolds, as then the model manifold and the support of the true distribution might not intersect [3]. This is the reason why many generative models add an additional noise term to the model distribution, resulting in blurry, degraded images. Instead of estimating the density p_R, *GANs* work by passing a random variable $\boldsymbol{z}$ through a parametric function G, which is typically implemented as a neural network. Therefore, it is possible to directly sample from the model distribution p_G, and by changing the model parameters $\boldsymbol{\theta_G}$, it is possible to adjust p_G to be similar to the real distribution p_R.

The main motivation behind *WGAN* is to analyze the different ways to measure distances between the model distribution p_G and the real distribution p_R. In the original formulation for *GANs* introduced by Goodfellow et al., they show that the objective function to optimize is the Jensen-Shannon (JS) divergence. For *WGAN*, the distance function used is the Earth-Mover (EM) distance or Wasserstein-1 distance, which intuitively computes the cost of the optimal transport plan to transform the distribution p_R into p_G. The Wasserstein-1 distance is defined as

$$W(p_R, p_G) = \inf_{\gamma \in \Pi(p_R, p_G)} \mathbb{E}_{(x,y)\sim\gamma}\left[\,||x-y||\,\right], \tag{3.7}$$

where $\Pi(p_R, p_G)$ is the set of all joint distributions (the set of all 'transport plans') $\gamma(x, y)$, whose marginals are p_R and p_G, respectively. As the infimum in Equation 3.7 is intractible, the dual form of the Wasserstein-1 distance [73] is used as a base to derive the optimization scheme

$$W(p_R, p_G) = \sup_{||D||_L \leq 1} \mathbb{E}_{\boldsymbol{x}\sim p_R}\left[D(\boldsymbol{x})\right] - \mathbb{E}_{\boldsymbol{x}\sim p_G}\left[D(\boldsymbol{x})\right], \tag{3.8}$$

where the supremum is over all 1-Lipschitz functions. By replacing this constraint with a K-Lipschitz assumption, the dual formulation only changes by a multiplicative constant of K, which does not change the optimization problem, as K can just be absorbed into the learning rate hyperparameter. If a parameterized family of functions D with parameters $\boldsymbol{\theta_D} \in W$ contains functions that are all K-Lipschitz, the following optimization problem, which leads to the main training objective for *WGAN*, can be derived. By assuming that the supremum in Equation 3.8 is obtained for some $\boldsymbol{\theta_D} \in W$ and by incorporating the generator network $G(\boldsymbol{z})$ as the model distribution p_G, we can reformulate the equation to arrive at

$$\max_{\boldsymbol{\theta_D}\in W} \mathbb{E}_{\boldsymbol{x}\sim p_R}\left[D(\boldsymbol{x})\right] - \mathbb{E}_{\boldsymbol{z}\sim p_z}\left[D\left(G(\boldsymbol{z})\right)\right]. \tag{3.9}$$

This formulation can be roughly approximated by training a neural network D, which is now called the *critic* instead of the discriminator. The critic is parameterized with weights $\boldsymbol{\theta_D}$ lying in a compact space W, and by backpropagating through $\mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}\left[\nabla_{\boldsymbol{\theta_G}} D\left(G(\boldsymbol{z})\right)\right]$, we arrive at the training scheme, which is similar to that of the original *GAN*. To have the critic weights $\boldsymbol{\theta_D}$ lie in a compact space, Arjovsky et al. suggest to clip the critic weights to a fixed box, with suggested parameters of $W = [-0.01, 0.01]$, after each gradient update. The full training procedure of *WGAN* is described in Algorithm 3.

When optimized, the Wasserstein-1 distance shows nicer properties than the *JS* divergence, such as a more sensible behavior when learning distributions supported by low dimensional manifolds. This results in *WGAN* being able to learn probability distributions where other learning objectives, such as those derived by the *JS* and *KL* divergences fail. Since the *EM* distance is continuous and differentiable, Arjovsky et al. suggest to train the critic to optimality, as the better trained critic will provide more reliable gradients. With the standard *JS* objective of the original *GAN*, this is not possible, since when the discriminator gets better, the *JS* distance is locally saturated, leading to the problem of vanishing gradients [2], while *WGAN* still has gradients everywhere. Arjovsky et al. also claim that *WGAN* resolves the issue of mode collapse due to the critic being trained until optimality. Additionally, an optimal critic allows for a meaningful interpretation of the

Algorithm 3: Training algorithm for *WGAN*, using minibatch stochastic gradient descent. Default suggested parameters are $\alpha = 0.00005$ for the learning rate, $c = 0.01$ for the clipping range and $k_{critic} = 5$ critic updates.

```
for number of training iterations do
    for $k_{critic}$ critic update steps do
        Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
        Take a minibatch $\boldsymbol{x}$ of $m$ images from the real data;
        Compute the stochastic gradient $g_D$ of the critic $D$ with respect to the
          critic weights $\boldsymbol{\theta_D}$:
            $$g_D = \nabla_{\boldsymbol{\theta_D}} \left[ \frac{1}{m} \sum_{i=1}^{m} \left[ D\left(x^{(i)}\right) - D\left(G(z^{(i)})\right) \right] \right] \quad (3.10)$$
        Update the critic weights using RMSProp [25];
        Clip the weights to be inside the range of $[-c, c]$;
    Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
    Compute the stochastic gradient $g_G$ of the generator $G$ with respect to the
      generator weights $\boldsymbol{\theta_G}$:
        $$g_G = -\nabla_{\boldsymbol{\theta_G}} \frac{1}{m} \sum_{i=1}^{m} \left[ D\left(G(z^{(i)})\right) \right] \quad (3.11)$$
    Update the generator weights using RMSProp [25];
```

critic loss, as it approximates the *EM* distance.

3.3.1 Wasserstein Generative Adversarial Networks with Gradient Penalty

While *WGAN* improved the training of *GANs* significantly, some settings still led to low-quality samples or complete lack of convergence. Gulrajani et al. argue that these failure modes are due to the way the 1-Lipschitz constraint is enforced in the original *WGAN*. The weight clipping present in the original *WGAN* shows, for some scenarios, that the critic has problems finding the optimal functions, therefore underusing its capacity. Additionally, weight clipping can also lead to vanishing or exploding gradient norms, which results in unstable gradients, and therefore, unstable training [21]. Gulrajani et al. prove that the optimal critic in the *WGAN* framework has unit gradient norm almost everywhere under p_R and p_G. This motivated them to enforce the 1-Lipschitz constraint of the original *WGAN* differently - by using a soft penalty on the gradient norm of the critic, the gradient norm can be encouraged towards 1, which forms the basis of the Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) training procedure [21]. For this gradient norm, Gulrajani et al. sample uniformly from a distribution $p_{\hat{\boldsymbol{x}}}$ that is defined along the lines between pairs of samples from the real data distribution p_R and the model distribution p_G. Implementing this gradient penalty, the original *WGAN* loss function is modified to

$$L = \underbrace{\mathbb{E}_{\boldsymbol{x}\sim p_R}\left[D(\boldsymbol{x})\right] - \mathbb{E}_{\boldsymbol{z}\sim p_z}\left[D\left(G(\boldsymbol{z})\right)\right]}_{\textit{WGAN} \text{ critic loss}} + \underbrace{\lambda\,\mathbb{E}_{\hat{\boldsymbol{x}}\sim p_{\hat{\boldsymbol{x}}}}\left[(||\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})||_2 - 1)^2\right]}_{\textit{WGAN-GP} \text{ gradient penalty}}, \tag{3.12}$$

where λ describes the penalty coefficient, which Gulrajani et al. found to work well for a multitude of experiments if set to a value of $\lambda = 10$. The *WGAN-GP* training procedure is described in Algorithm 4. Compared to *WGAN* and *DCGAN*, which used batch normalization in both the generator and the critic/discriminator, *WGAN-GP* does not use batch normalization in the critic, as the gradient penalty term in the objective would not be valid [21], and it is suggested to use layer normalization [4] instead. Samples of a *WGAN-GP* trained on the LSUN [76] bedrooms dataset can be seen in Figure 3.7.

Algorithm 4: Training algorithm for *WGAN-GP*, using minibatch stochastic gradient descent. Suggested parameters are $\alpha = 0.0001$ for the learning rate, $\lambda = 10$ for the penalty coefficient, $\beta_1 = 0, \beta_2 = 0.9$ for the decay rates of the Adaptive Moment Estimation (Adam) [32] optimizer and $k_{critic} = 5$ critic updates.

```
for number of training iterations do
    for $k_{critic}$ critic update steps do
        Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
        Take a minibatch $\boldsymbol{x}$ of $m$ images from the real data;
        Compute random interpolates between real and generated data, using uniform
          noise $\epsilon \sim U[0,1]$
              $$\hat{\boldsymbol{x}} = \epsilon\,\boldsymbol{x} + (1-\epsilon)\hat{\boldsymbol{x}} \tag{3.13}$$
          Compute the gradient $g_D$ of the critic $D$ with respect to the critic weights $\boldsymbol{\theta_D}$:
              $$g_D = \nabla_{\boldsymbol{\theta_D}}\left[\frac{1}{m}\sum_{i=1}^{m}\left[D\left(x^{(i)}\right) - D\left(G(z^{(i)})\right) + \lambda\left[\left(\left|\left|\nabla_{\hat{x}^{(i)}} D(\hat{x}^{(i)})\right|\right|_2 - 1\right)^2\right]\right]\right] \tag{3.14}$$
        Update the critic weights using Adam [32];
    Generate a minibatch $G(\boldsymbol{z})$ of $m$ images from the Generator $G$;
    Compute the gradient $g_G$ of the generator $G$ with respect to the generator weights $\boldsymbol{\theta_G}$:
              $$g_G = -\nabla_{\boldsymbol{\theta_G}}\frac{1}{m}\sum_{i=1}^{m}\left[D\left(G(z^{(i)})\right)\right] \tag{3.15}$$
    Update the generator weights using Adam [32];
```

The improved training stability, especially on multiple datasets, and the solid theoretical foundation based on *WGAN* was the main reasoning for us to choose *WGAN-GP* as the base of our implementation described in Chapter 4.

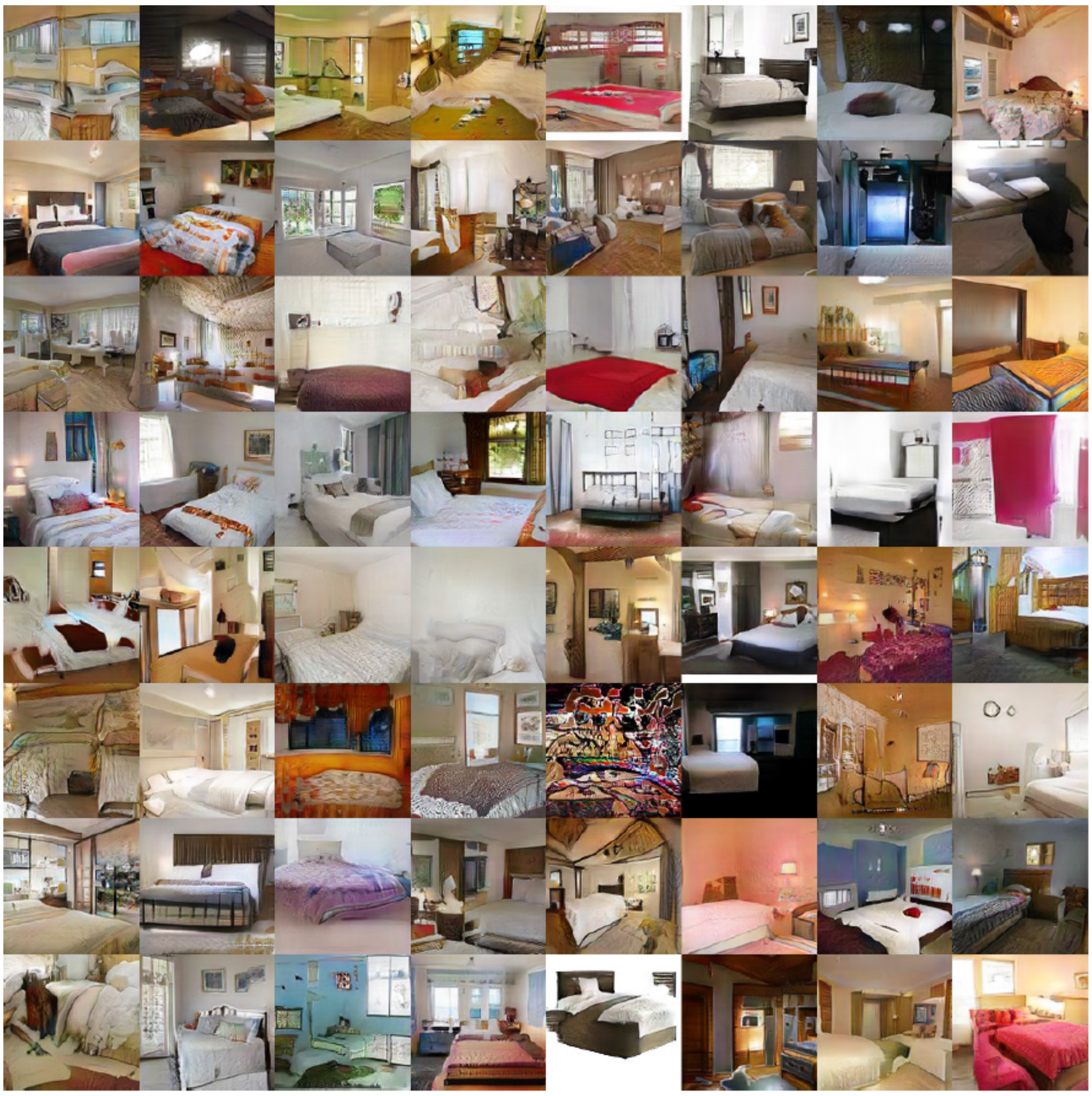


Figure 3.7: Samples from *WGAN-GP* after convergence when trained on LSUN [76] bedrooms. Image taken from [21].

3.4 SimGAN

As an application of *GANs* closely related to data augmentation, and therefore to our contribution, we will discuss *SimGAN* [64] in this section. The main idea of *SimGAN* is to render simulated image data, for which the annotation is known, and to incorporate a *GAN* to refine the synthetic data. The goal of this refinement process is to use real, unannotated data to improve the realism of the rendered image data, while preserving the annotation information [64]. Without requiring human annotation effort, Shrivastava et al. achieved state-of-the-art classification results using their SimGAN architecture. The overview of the SimGAN architecture is illustrated in Figure 3.8.

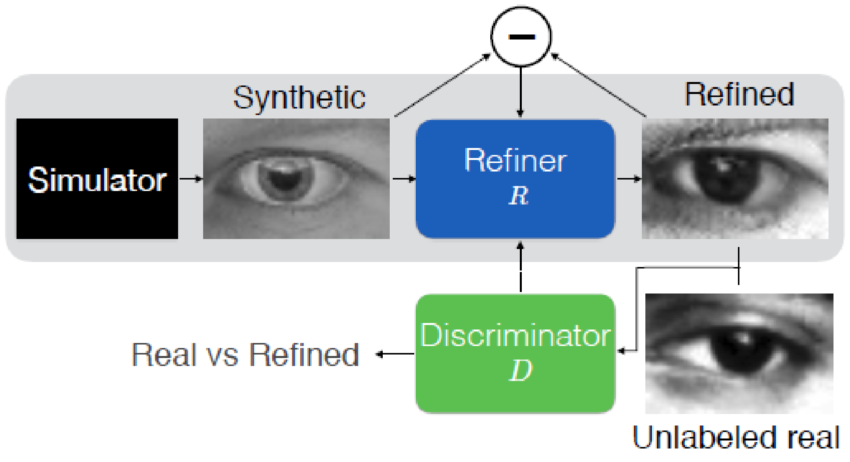


Figure 3.8: SimGAN architecture for unsupervised synthetic image refinement. Image taken from [64].

SimGAN uses a number of modifications compared to the traditional *GAN* formulation. In addition to the standard adversarial loss, they introduced a *self-regularization* loss that minimizes a feature transform of the synthetic and real images $l_{reg} = ||\psi(x_{ref}) - x_{syn}||_1$, where ψ is a mapping from image space to a feature space. In their paper, Shrivastava et al. simply used an identity mapping ($\psi(x) = x$) as the default transform. This self-regularization loss forces the refinement network to produce refined images that resemble the unlabeled real images more closely, while still preserving the annotation information of the rendered image [64]. The second major modification of SimGAN is that it uses a *local adversarial loss*. Key motivation for this approach is that the refiner network tends to over-emphasize certain image features to fool the current discriminator network, which can lead to artifacts [64]. To prevent this, Shrivastava et al. design the discriminator such that it outputs a probability map of size $[w \times h]$, effectively splitting the input image into $[w \times h]$ patches. The adversarial loss function is then simply computed as the sum of the cross-entropy losses over the local patches. This helps the refinement network to produce images, where every local patch has similar statistics to a real image patch. The local adversarial loss approach is illustrated in Figure 3.9. In order to further stabilize training and reduce artifacts, Shrivastava et al. also implemented a history of refined images. Instead of sampling a new minibatch of images as the input every time, SimGAN keeps a buffer of B images, and takes half of the images of a minibatch from this buffer. After every iteration, half of the refined images randomly replace previous images inside this buffer. By introducing memory into the discriminator, this improves the consistency of training [64]. This

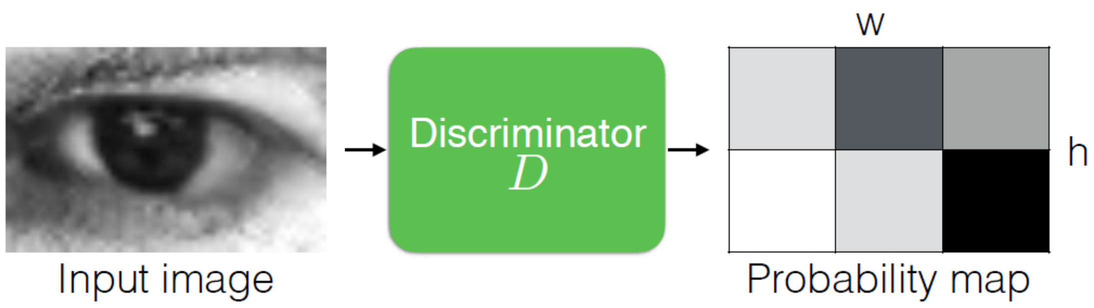


Figure 3.9: SimGAN local adversarial loss. Image taken from [64].

concept is illustrated in Figure 3.10.

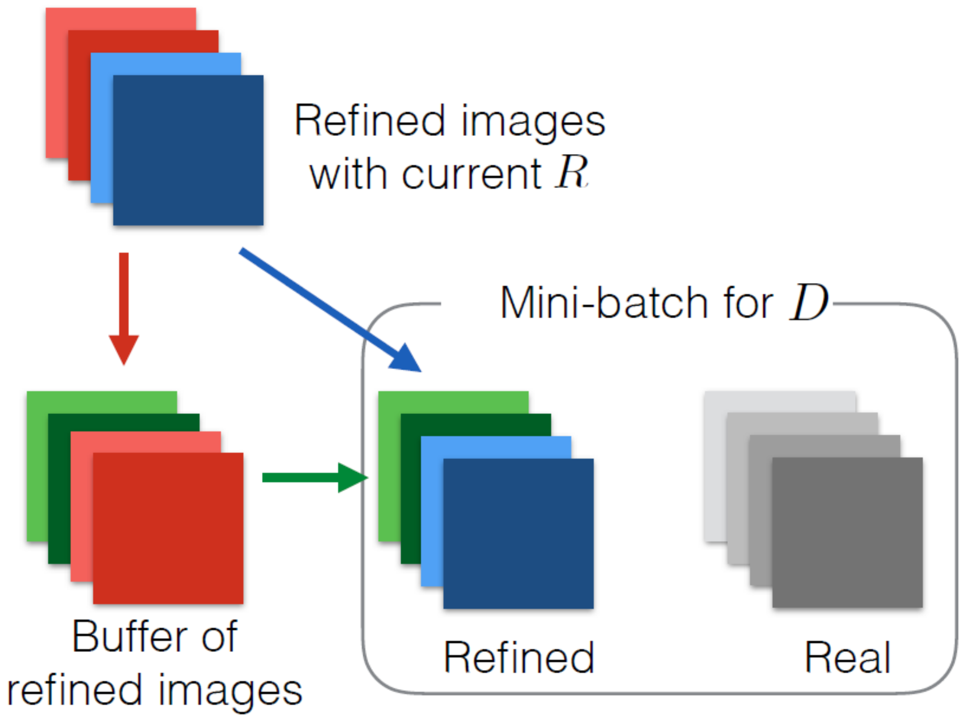


Figure 3.10: SimGAN history of refined images to improve stability. Image taken from [64].

SimGAN shows that *GANs* have potential to be used for data augmentation, which is the main topic of our investigation in Chapter 4.

3.5 Summary

In this chapter, the major milestones in *GAN* research were presented. Introduced very recently, the original *GAN* completely changed modern generative modeling. Being able to implicitly estimate a model distribution using just data in combination with popular deep learning methods such as backpropagation and *SGD* made for a very attractive generative model. However, early *GANs* struggled with stability, often resulting in failures during training. Motivated by this idea, and especially the potential application for unsupervised feature extraction, the *DCGAN* architecture was introduced, which resulted in a much more stable training procedure when following the *DCGAN* guidelines. Finally, with the advent of research on the theoretical formulation of *GANs*, *WGAN* and *WGAN-GP* were introduced, which fundamentally changed the objective function of *GAN* to include the *EM* distance, resulting in stable training with a solid theoretical foundation. *GANs* definitely show potential in a lot of applications, which is where most research is focused on to this date. Especially related to our investigation, applications such as SimGAN show that *GANs* also have potential for use in the domain of data augmentation. In Chapter 4, we will discuss our approach for using *GANs* in a way to synthesize additional labeled training data for image segmentation.

4 Data Augmentation Using Generative Adversarial Networks

Contents

4.1 Modifying Generative Adversarial Networks for Data Augmentation 57
4.2 Implementation Details 59
4.3 Evaluation Metrics 64

In this chapter, we present our novel modification to Generative Adversarial Networks (GANs), which we use to synthesize images and corresponding segmentation masks at the same time. We describe how to arrive at our *GANs* architecture, and evaluate the generated images and segmentation masks by using them as training data for two image segmentation tasks. Additionally, we compare the *GAN*-based data augmentation to standard data augmentation techniques, such as described in Section 2.7. In Section 5.1, we evaluate on the *SCR Lung Database* [72], a small, medical image segmentation dataset. For the second evaluation shown in Section 5.2, we chose the *Cityscapes* [10] dataset (see Figure 5.6 for an example image and segmentation mask) for semantic segmentation in order to test our approach on a larger, challenging dataset.

4.1 Modifying Generative Adversarial Networks for Data Augmentation

As discussed in Chapter 3, *GANs* are the current state-of-the-art method to learn generative models. However, the standard *GAN* definition only allows for the generation of images, without respective labels. Therefore, for the generated data to be used for data augmentation, the conventional *GAN* formulation needs to be modified to also generate corresponding labels. During the course of this thesis, we proposed a novel *GAN* architecture, which jointly generates images and their corresponding segmentation masks, for direct use of training data augmentation [45]. Compared to the standard *GAN* formulation, this architectural adaptation is straight-forward to implement, as it is a simple change in the network architecture, and can therefore be used with any *GAN* training scheme, such as the Wasserstein Generative Adversarial Network (WGAN) or the more recent Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP).

The main idea of our proposed architecture is that we fuse the image and segmentation mask to create an *image-segmentation pair*. This is done by concatenating both images along the channel axis. As an example, a given RGB image of dimensions $[W \times H \times 3]$ with its corresponding segmentation mask of dimensions $[W \times H \times 1]$ results in an image-segmentation pair with dimensions of $[W \times H \times 4]$. When training the *GAN*, the generator is now modified to generate image-segmentation pairs, instead of just images. This change, in its most trivial form, can be achieved by simply modifying the final convolutional layer in the generator, such that the number of output channels is equal to the number of channels of the required image-segmentation pair. For the discriminator, a similar principle is applied. The discriminator now takes image-segmentation pairs as input, and its goal is to correctly decide if any given image-segmentation pair is real or synthetic. Therefore, the first convolutional layer of the discriminator needs to be modified to accept inputs where the number of channels is equal to the number of channels of the image-segmentation pair. Our *GAN* architecture that we use to generate image-segmentation pairs is illustrated in Figure 4.1.

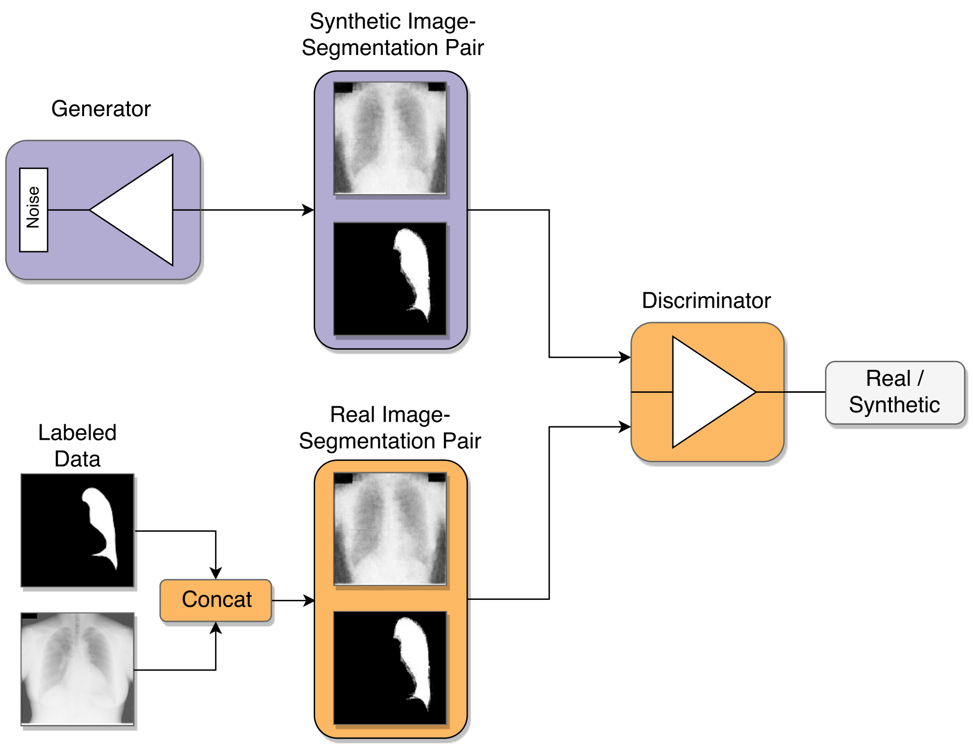


Figure 4.1: *GAN* architecture for generation of image-segmentation pairs.

While we showed some preliminary results for this architecture in [45], for this thesis, we perform a much more extensive evaluation to find out how well *GANs* perform for data augmentation, especially when compared to standard data augmentation. For all of our evaluations, we define *'standard'* data augmentation as a combination of image transformations, such as described in Section 2.7, while *GAN-based* augmentation refers to adding synthetic image-segmentation pairs from our modified *GAN* architecture to the training set of a segmentation network.

4.2 Implementation Details

All *GANs* in our experiments are trained using the *WGAN-GP* training scheme and loss function, as we found this to be the most robust method for training *GANs*, even across multiple datasets. For the gradient penalty hyperparameter, we used the default value suggested by Gulrajani et al., setting $\lambda = 10$. Additionally, compared to [45], we significantly increased the image resolution, in order to test how well the *GAN* is able to handle higher resolutions. Our code is based on the code provided by the authors of *WGAN-GP* [21][1], using the TensorFlow [1] deep learning framework. For both our evaluations, we use a similar evaluation setup. First, we split the data into one or multiple training, validation, and test sets. Then, we train *GANs* for every training set of this dataset, until the image quality does not further improve. Finally, we take the fully-trained generator network, and use it directly as an input to a U-Net [54] style segmentation network. Compared to our previous evaluation [45], where we simply sampled a fixed amount of image-segmentation pairs, this allows for the full range of possible images to be sampled from the generator, capturing the full amount of variation that the generator has learned from the data. For training the segmentation network, we use different ratios of real and generated data, and apply either no additional standard data augmentation, or a combination of standard data augmentation methods. When mixing real and generated data, we exclusively use only the specific *GAN* that was trained on the same real training data set, to keep all training sets separate. Our evaluation setup is illustrated in Figure 4.2. Additionally, for all segmentation networks, early stopping is used. For every segmentation network, we train for at least 3000 iterations. During training, we keep track of the minimal loss and its iteration number, as well as the network parameters at the loss minimum, measured for the internal validation set. We only stop the training when 3000 iterations have passed since the last validation loss minimum was found. In practice, this resulted in a good compromise of network performance and training time. The final metric for our evaluation is the segmentation performance of the segmentation network on the unseen test data. For the evaluation on the test set, and therefore the final segmentation performance, we upsample the output of our segmentation networks using bicubic upsampling to the target resolution. When computing the final segmentation masks of the SCR Lung Database,

[1]GitHub: Improved Training of Wasserstein *GANs*, `https://github.com/igul222/improved_wgan_training`, Accessed: 01.02.2018

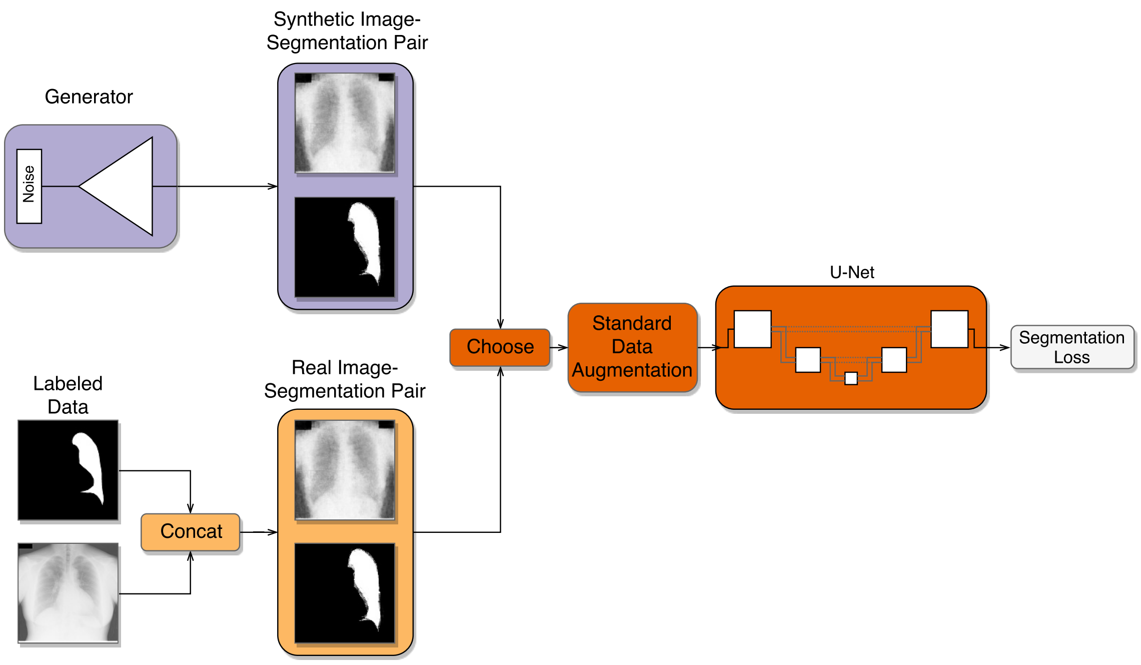

Synthetic Image-Segmentation Pair
Generator
Noise
U-Net
Choose
Standard Data Augmentation
Segmentation Loss
Labeled Data
Real Image-Segmentation Pair
Concat

Figure 4.2: Evaluation setup containing a pre-trained generator and U-Net style segmentation network.

we additionally apply largest-connected-component post-processing, in order to only extract the single largest foreground object, as we are only interested in a single object (the left lung) in this evaluation. Afterwards, we compute the Dice coefficient, the Hausdorff distance (for the SCR Lung Database), and the mean Intersection-over-Union (mIoU) (for the Cityscapes dataset) as our evaluation metrics. All evaluations were done on an NVIDIA Tesla K80 with 12GB of GPU memory, although all networks were designed for an NVIDIA GTX980M with 8GB of GPU memory. For both evaluations, all input images were intensity-normalized to a range of $[-1, 1]$.

4.2.1 Network Architectures: SCR Lung Database

Our *GAN* architecture for the SCR Lung Database is based on the Deep Convolutional Generative Adversarial Network (DCGAN) [52], with added layers to handle the higher resolution and added modifications for handling image-segmentation pairs. Even though *WGAN-GP* has no issues when training on more complex architectures, such as ResNet [23]-based architectures, due to the required time when training a *GAN*, we decided to stick to a simpler architecture. All architectural parameters of our *GAN* for the SCR Lung Database are listed in Table 4.1. Our segmentation network is based on U-Net [54], with a depth of 3. As described in Section 2.6.3, we use the sigmoid cross-entropy loss for computing our segmentation loss. Additionally, we used a constant number of 64 kernels in every convolutional layer of our segmentation network, as we found that to result in the most consistent performance. The architectural parameters of our segmentation network for this evaluation are listed in Table 4.2.

SCR Lung Database Generator Architecture

Minibatch Size: 16, Optimizer: Adaptive Moment Estimation (Adam) ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$).
All weights were initialized using the *He* initializer [22].
The input noise vector of dimensionality 128 was drawn from a zero-mean Gaussian distribution with unit variance.

	Input Projection	Generator Stage 1	Generator Stage 2	Generator Stage 3	Generator Stage 4	Generator Stage 5	Generator Stage 6
Type	Fully-Connected	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution
Input Dimension	$[1 \times 128]$	$[4 \times 4 \times 2048]$	$[8 \times 8 \times 1024]$	$[16 \times 16 \times 512]$	$[32 \times 32 \times 256]$	$[64 \times 64 \times 128]$	$[128 \times 128 \times 64]$
Output Dimension	$[4 \times 4 \times 2048]$	$[8 \times 8 \times 1024]$	$[16 \times 16 \times 512]$	$[32 \times 32 \times 256]$	$[64 \times 64 \times 128]$	$[128 \times 128 \times 64]$	$[256 \times 256 \times 2]$
Number of Kernels	-	1024	512	256	128	64	2
Kernel Size	-	5	5	5	5	5	5
Stride	-	2	2	2	2	2	2
Padding	-	1	1	1	1	1	1
Activation	Rectified Linear Unit (ReLU)	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	Hyperbolic Tangent (tanh)
Batch Normalization	yes	yes	yes	yes	yes	yes	no

(a) Generator architecture of our *GAN* trained on the SCR Lung Database.

SCR Lung Database Discriminator Architecture

Minibatch Size: 16, Optimizer: *Adam* ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$).
All weights were initialized using the *He* initializer [22].

	Discriminator Stage 1	Discriminator Stage 2	Discriminator Stage 3	Discriminator Stage 4	Discriminator Stage 5	Discriminator Stage 6	Output
Type	Convolution	Convolution	Convolution	Convolution	Convolution	Convolution	Fully-Connected
Input Dimension	$[256 \times 256 \times 2]$	$[128 \times 128 \times 32]$	$[64 \times 64 \times 64]$	$[32 \times 32 \times 128]$	$[16 \times 16 \times 256]$	$[8 \times 8 \times 512]$	$[4 \times 4 \times 1024]$
Output Dimension	$[128 \times 128 \times 32]$	$[64 \times 64 \times 64]$	$[32 \times 32 \times 128]$	$[16 \times 16 \times 256]$	$[8 \times 8 \times 512]$	$[4 \times 4 \times 1024]$	$[1]$
Number of Kernels	32	64	128	256	512	1024	-
Kernel Size	5	5	5	5	5	5	-
Stride	2	2	2	2	2	2	-
Padding	2	2	2	2	2	2	-
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	-
Batch Normalization	yes	yes	yes	yes	yes	yes	no

(b) Discriminator architecture of our *GAN* trained on the SCR Lung Database.

Table 4.1: Architecture of the *GAN* trained on the SCR Lung Database.

SCR Lung Database Segmentation Network Architecture

Minibatch Size: 16, Optimizer: *Adam* ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$).
All weights were initialized using the *He* initializer [22].
No Batch-Normalization was used in any layer.

	Encoder Stage 1	Encoder Stage 2	Encoder Stage 3	Encoded Features	Decoder Stage 1	Decoder Stage 2	Decoder Stage 3	Output
Type	Convolution Convolution Pooling	Convolution Convolution Pooling	Convolution Convolution Pooling	Convolution Convolution	Upsampling Concat (Encoder 3) Convolution Convolution	Upsampling Concat (Encoder 2) Convolution Convolution	Upsampling Concat (Encoder 1) Convolution Convolution	Convolution
Input Dimension	$[256 \times 256 \times 1]$	$[128 \times 128 \times 64]$	$[64 \times 64 \times 64]$	$[32 \times 32 \times 64]$	$[32 \times 32 \times 64]$	$[64 \times 64 \times 64]$	$[128 \times 128 \times 64]$	$[256 \times 256 \times 64]$
Output Dimension	$[128 \times 128 \times 64]$	$[64 \times 64 \times 64]$	$[32 \times 32 \times 64]$	$[32 \times 32 \times 64]$	$[64 \times 64 \times 64]$	$[128 \times 128 \times 64]$	$[256 \times 256 \times 64]$	$[256 \times 256 \times 2]$
Number of Kernels	64	64	64	64	64	64	64	2
Kernel Size	3	3	3	3	3	3	3	3
Stride	1	1	1	1	1	1	1	1
Padding	1	1	1	1	1	1	1	1
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*
Pooling	$[2 \times 2]$, Average	$[2 \times 2]$, Average	$[2 \times 2]$, Average	-	-	-	-	

Table 4.2: Architecture of the U-Net based segmentation network trained on the SCR Lung Database.

4.2.2 Network Architectures: Cityscapes

Similar to Section 4.2.1, our *GAN* is based on *DCGAN* [52], and our segmentation network is based on U-Net [54]. Our *GAN* mostly follows the same architecture as before, and is only slightly adjusted for the different image size and image depth, due to the images being RGB. The detailed listing of all architectural parameters of our *GAN* trained on the Cityscapes dataset is shown in Table 4.3. For our segmentation network, compared to our evaluation on the SCR Lung Database, we increased the number of filters for all convolutional layers to 256, as well as the depth of the network to 4, as the semantic segmentation of the Cityscapes dataset is a much more difficult problem, therefore requiring a more powerful segmentation network. The architecture of the segmentation network used for the Cityscapes dataset is shown in more detail in Table 4.4.

Cityscapes Generator Architecture						
Minibatch Size: 8, Optimizer: *Adam* ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$). All weights were initialized using the *He* initializer [22]. The input noise vector of dimensionality 128 was drawn from a zero-mean Gaussian distribution with unit variance.						
	Input Projection	Generator Stage 1	Generator Stage 2	Generator Stage 3	Generator Stage 4	Generator Stage 5
Type	Fully-Connected	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution	Fractionally Strided Convolution
Input Dimension	$[1 \times 128]$	$[8 \times 4 \times 1024]$	$[8 \times 16 \times 512]$	$[32 \times 16 \times 256]$	$[64 \times 32 \times 128]$	$[128 \times 64 \times 64]$
Output Dimension	$[8 \times 4 \times 1024]$	$[16 \times 8 \times 512]$	$[32 \times 16 \times 256]$	$[64 \times 32 \times 128]$	$[128 \times 64 \times 64]$	$[256 \times 128 \times 4]$
Number of Kernels	-	512	256	128	64	4
Kernel Size	-	5	5	5	5	5
Stride	-	2	2	2	2	2
Padding	-	1	1	1	1	1
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*tanh*
Batch Normalization	yes	yes	yes	yes	yes	no

(a) Generator architecture of our *GAN* trained on the Cityscapes dataset.

Cityscapes Discriminator Architecture						
Minibatch Size: 8, Optimizer: *Adam* ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$). All weights were initialized using the *He* initializer [22].						
	Discriminator Stage 1	Discriminator Stage 2	Discriminator Stage 3	Discriminator Stage 4	Discriminator Stage 5	Output
Type	Convolution	Convolution	Convolution	Convolution	Convolution	Fully-Connected
Input Dimension	$[256 \times 128 \times 4]$	$[128 \times 64 \times 32]$	$[64 \times 32 \times 64]$	$[32 \times 16 \times 128]$	$[16 \times 8 \times 256]$	$[8 \times 4 \times 512]$
Output Dimension	$[128 \times 64 \times 32]$	$[64 \times 32 \times 64]$	$[32 \times 16 \times 128]$	$[16 \times 8 \times 256]$	$[8 \times 4 \times 512]$	$[1]$
Number of Kernels	32	64	128	256	512	-
Kernel Size	5	5	5	5	5	-
Stride	2	2	2	2	2	-
Padding	2	2	2	2	2	-
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	-
Batch Normalization	yes	yes	yes	yes	yes	no

(b) Discriminator architecture of our *GAN* trained on the Cityscapes dataset.

Table 4.3: Architecture of the *GAN* trained on the Cityscapes dataset.

Cityscapes Segmentation Network Architecture						
Minibatch Size: 8, Optimizer: *Adam* ($\eta = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.9$). All weights were initialized using the *He* initializer [22]. No Batch-Normalization was used in any layer.						
	Encoder Stages 1/2	Encoder Stages 3/4	Encoded Features	Decoder Stages 1/2	Decoder Stages 3/4	Output
Type	Convolution Convolution Pooling Convolution Convolution Pooling	Convolution Convolution Pooling Convolution Convolution Pooling	Convolution Convolution	Upsampling Concat (Encoder 4) Convolution Convolution Upsampling Concat (Encoder 3) Convolution Convolution	Upsampling Concat (Encoder 2) Convolution Convolution Upsampling Concat (Encoder 1) Convolution Convolution	Convolution
Input Dimension	$[256{\times}256{\times}3]$	$[64 \times 64 \times 256]$	$[16 \times 16 \times 256]$	$[16 \times 16 \times 256]$	$[64 \times 64 \times 256]$	$[256 \times 256 \times 256]$
Output Dimension	$[64{\times}64{\times}256]$	$[16 \times 16 \times 256]$	$[16 \times 16 \times 256]$	$[64 \times 64 \times 256]$	$[256 \times 256 \times 256]$	$[256 \times 256 \times 8]$
Number of Kernels	256	256	256	256	256	8
Kernel Size	3	3	3	3	3	3
Stride	1	1	1	1	1	1
Padding	1	1	1	1	1	1
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*
Pooling	$[2 \times 2]$, Average	$[2 \times 2]$, Average	-	-	-	

Table 4.4: Architecture of the U-Net style segmentation network used for the Cityscapes dataset.

4.3 Evaluation Metrics

In this section, we briefly describe the evaluation metrics that we use in our experiments.

4.3.1 Dice Coefficient

The Sørenson-Dice coefficient, which nowadays is most commonly referred to as the Dice coefficient or Dice score, is a metric for computing the similarity between two samples ([12],[65]). It is computed as

$$DSC = \frac{2TP}{2TP + FP + FN}, \tag{4.1}$$

where TP, FP and FN describe the *true positives*, *false positives* and *false negatives* between both samples, respectively. It can easily be applied to the problem of image segmentation, by simply computing a confusion matrix between predicted and labeled segmentation mask, and deriving the components of the Dice score from the confusion matrix.

4.3.2 Intersection-over-Union

The Intersection-over-Union (IoU), also called *Jaccard Index* is very similar to the Dice coefficient, and is also used to compute a similarity measure between two samples. It is computed as

$$IoU = \frac{TP}{TP + FP + FN}, \tag{4.2}$$

which simply uses a different weighting for true positives, compared to the Dice coefficient.

4.3.3 Hausdorff Distance

The *Hausdorff distance* measures the distance between two sets, by measuring how close each point of one set is to all points of the other set. The maximum distance (i.e. the farthest distance from a point in one set to its nearest neighbor in the other set) is the Hausdorff distance [28]. It is defined as

$$H(A, B) = \max(h(A, B), h(B, A)), \tag{4.3}$$

where

$$h(A, B) = \max_{a \in A} \min_{b \in B} ||a - b|| \tag{4.4}$$

is the directed Hausdorff distance, and $||a - b||$ is a norm (e.g. the Euclidean norm) between point a from set A and point b from set B. Intuitively, the Hausdorff distance describes the most *mismatched* point between set A and set B, and therefore describes the largest distance between any points of A and B [28]. Unlike the Dice and IoU, the Hausdorff distance has no explicit pairing of points, i.e. many points of A might be close to a single point of B.

5 Evaluation and Results

Contents

5.1 SCR Lung Database Medical Image Segmentation 67

5.2 Cityscapes Semantic Image Segmentation 73

In this chapter, we describe both datasets used for our evaluation as well as present our evaluation setup for every dataset. Finally, we present qualitative and quantitative results for both the SCR Lung Database as well as the Cityscapes dataset, and discuss these results in detail.

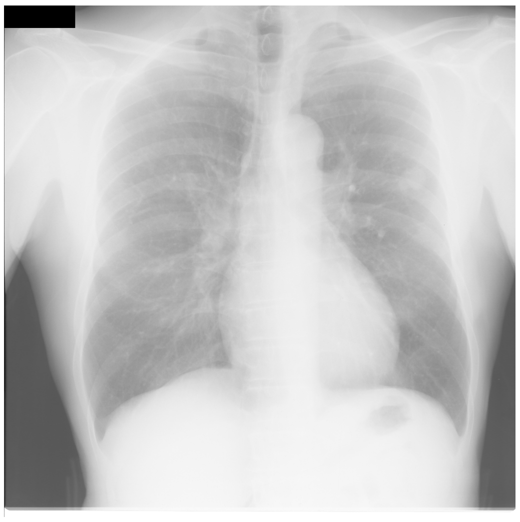
5.1 SCR Lung Database Medical Image Segmentation

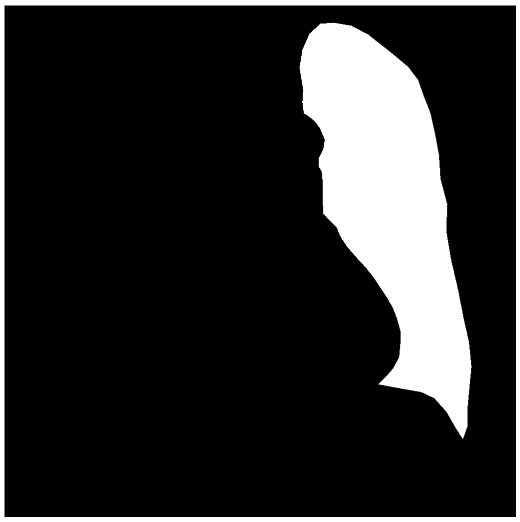
5.1.1 Dataset Description

The *SCR Lung Database* [72] is a dataset consisting of 247 chest X-ray images, taken from the JSRT database [63]. Its image resolution is $[2048 \times 2048]$ at a physical resolution of 0.175 mm per pixel in each dimension, and it contains groundtruth segmentation masks for 5 objects: both lungs, the heart and both clavicles. Additionally, 154 images contain exactly one pulmonary lung nodule each, while the other 93 images contain none. For our evaluation, we chose the task of segmenting the left lung from the image. An example image with its corresponding segmentation mask can be seen in Figure 5.1.

5.1.2 Evaluation Procedure

All images are downsampled to a resolution of $[256 \times 256]$ before we use them for training in order to fit all our networks into GPU memory while still being able to use a large enough minibatch size for stable training. We shuffle the dataset randomly and split it into 3 folds, each containing 135 training images, 30 validation images and 82/83 test images, chosen such that all images are contained exactly once in the set of test images.


(a) Example image from the SCR Lung Database.


(b) Example segmentation mask from the SCR Lung Database. Segmentation classes are listed in the legend below.

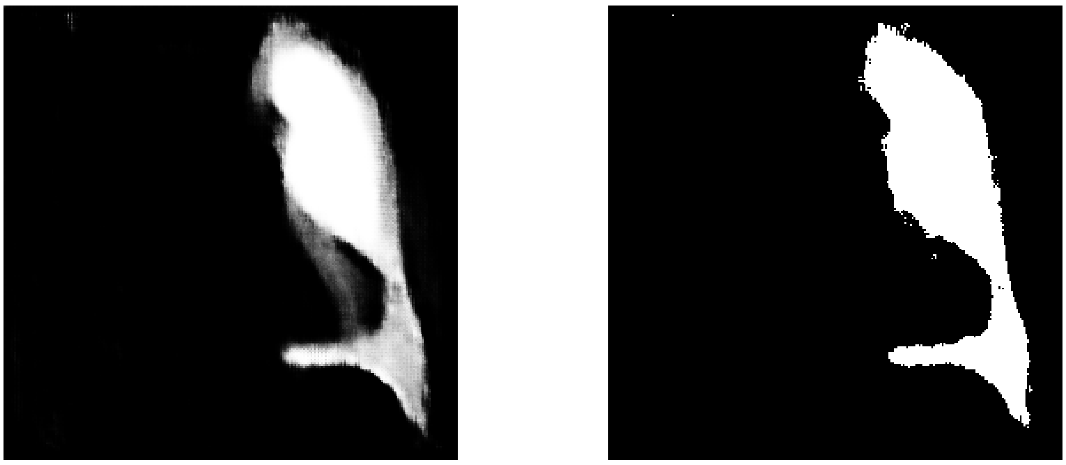
Classes:
■ Background □ Foreground (Left Lung)

Figure 5.1: Example image and category segmentation mask from the SCR Lung Database.

For the final evaluation of the segmentation performance, we report performance as the average Dice score and Hausdorff distance over all folds.

As the first step of our evaluation, we train a Generative Adversarial Network (GAN) with our modified architecture (see Section 4.1) for each of the 3 folds of training data, resulting in 3 fully-trained *GANs*. As it is still difficult to determine a quantifiable stopping criterion for the training of *GANs*, every *GAN* was trained for 10000 iterations, which took approximately 24 hours per *GAN*. The raw image-segmentation pairs from the generator are in the intensity range of $[-1, 1]$. Therefore, when training our segmentation network using generated images, we threshold all segmentation masks at 0 when computing the segmentation loss. An example for a segmentation mask sampled from the *GAN* before and after thresholding can be seen in Figure 5.2.

For the main part of our evaluation of the SCR Lung Database, we train multiple segmentation networks for every fold, using an exhaustive set of combinations of real and generated data as well as with and without standard data augmentation. We evaluated different combinations of standard data augmentation on one fold of the validation set to find suitable augmentation parameters for the final comparison. For this parameter search, we fixed elastic deformation at 10 pixels for each control point, to speed up the search. These results for different augmentation methods are shown in Table 5.1, and the combination of parameters listed in bold will be used as our standard data augmentation method for the final evaluation. However, we also tried several other combinations of



(a) Raw segmentation image sampled from the *GAN* in the intensity range of $[-1, 1]$.

(b) Binary segmentation mask after thresholding the *GAN* output at 0.

Figure 5.2: Raw output and thresholded segmentation mask sampled from the *GAN* trained on the SCR Lung Database. Segmentation classes in the thresholded image are the same as in Figure 5.1.

augmentation parameters and found the difference in validation performance to be negligible. Important to note is that this parameter search was done on only a single fold of the validation set, therefore those results are not comparable to our final qualitative results which are averaged over all folds. Before computing our final segmentation performance metrics, we remove everything from the resulting segmentation mask except for the largest connected component, as we only want to predict the left lung, which is a single connected component. Training each segmentation network took approximately 8 hours on our setup.

5.1.3 Results

The full details of our implementation and network architectures are described in Section 4.2. Sample images from our *GANs* trained on the SCR Lung Database can be seen in Figure 5.3. The final segmentation performance, averaged across all folds, is shown in Table 5.2. In order to better compare *GAN*-based data augmentation and standard data augmentation, we also present samples of resulting segmentation masks for two of our networks: the network trained on a mix of real and generated data without data augmentation (*GANs*-based augmentation), and the network trained solely on real data with standard data augmentation. Some of the best resulting samples are shown in Figure 5.4, while the example showing the worst performance is presented in Figure 5.5.

Augmentation Parameters				Validation Performance
Intensity shift around zero (stddev)	Intensity scaling around zero (stddev)	Random translation around zero (stddev)	Elastic deformation around zero (stddev)	Dice (mean)
-	-	-	-	0.9698
-	-	-	10 px	0.9706
-	-	10 px	10 px	0.9685
-	0.50	-	10 px	0.9703
-	0.50	10 px	10 px	0.9677
0.50	-	-	10 px	0.9640
0.50	-	10 px	10 px	0.9691
0.50	0.50	-	10 px	0.9663
0.50	**0.50**	**10 px**	**10 px**	**0.9712**

Table 5.1: Comparison of augmentation parameters for the SCR Lung Database. All results were computed by training strictly on real data and recording the best validation performance. For illustrative purposes, we only report the Dice score, although the Hausdorff distance and mean Intersection-over-Union (mIoU) score follow the same ordering of segmentation performance. For further evaluation, we use the augmentation parameters listed in **bold** as our standard data augmentation.

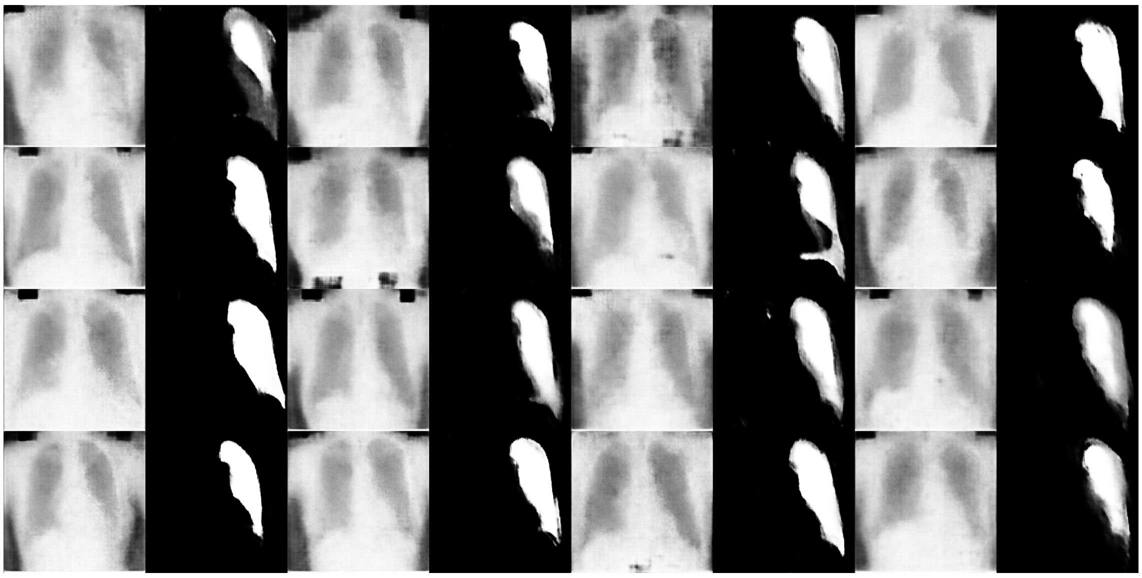


Figure 5.3: Sample images from our *GAN* trained on the SCR Lung Database. Odd columns show generated images, while even columns show the respective generated segmentation masks.

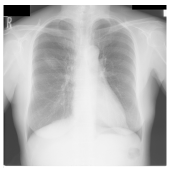
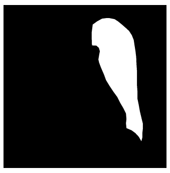
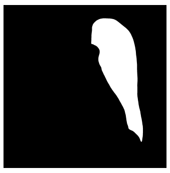
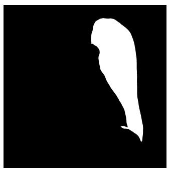
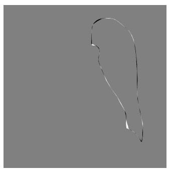
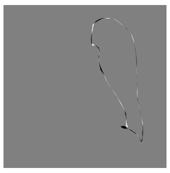
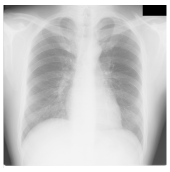
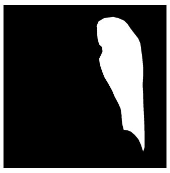
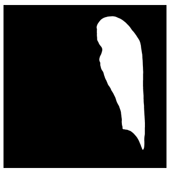
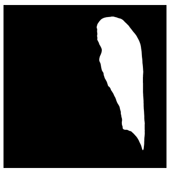
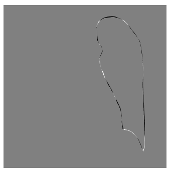
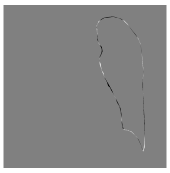
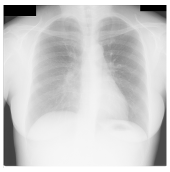
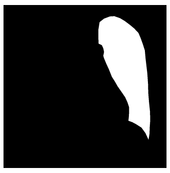
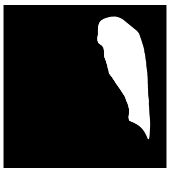
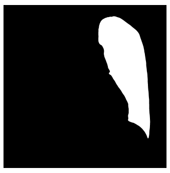
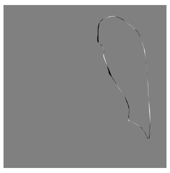
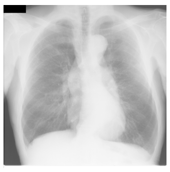
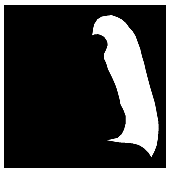
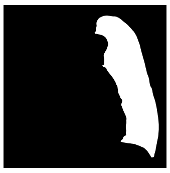
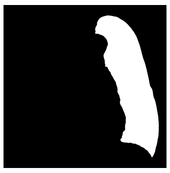
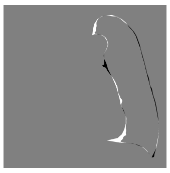
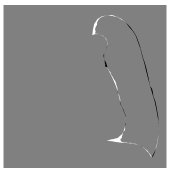
Image	Groundtruth	Prediction *GAN*-based Aug.	Prediction Standard Aug.	Difference: *GAN*-based Aug.	Difference: Standard Aug.
					
					
					
					

Figure 5.4: Comparison of good segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the SCR Lung Database. Columns 3 and 5 show results from our segmentation network trained using *GAN*-based augmentation with a mix of real and generated data, while Columns 4 and 6 show the results of our segmentation network trained using standard data augmentation.

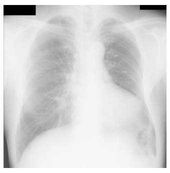
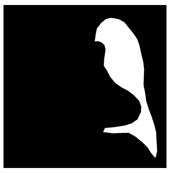
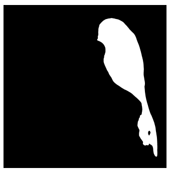
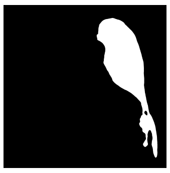
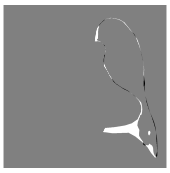
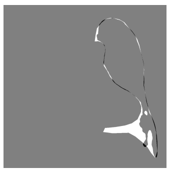
Image	Groundtruth	Prediction *GAN*-based Aug.	Prediction Standard Aug.	Difference: *GAN*-based Aug.	Difference: Standard Aug.
					

Figure 5.5: Comparison of the worst performing test sample from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the SCR Lung Database. Columns 3 and 5 show results from our segmentation network trained using *GAN*-based augmentation with a mix of real and generated data, while Columns 4 and 6 show the results of our segmentation network trained using standard data augmentation.

Method	Number of real pairs in minibatch	Number of generated pairs in minibatch	Aug.?	Dice (mean)	Dice (stddev)	Hausdorff (mean)	Hausdorff (stddev)
This Work	16	0	yes	0.9765	0.0165	1.2057 mm	0.3131 mm
	16	0	no	0.9742	0.0166	1.2626 mm	0.3440 mm
	8	8	yes	0.9765	0.0228	**1.1722 mm**	0.3313 mm
	8	8	no	**0.9768**	0.0147	1.2106 mm	0.3154 mm
	0	16	yes	0.9655	0.0163	1.2651 mm	0.3067 mm
	0	16	no	0.9632	0.0202	1.3273 mm	0.3434 mm
Previous Neff et al. [45]	16	0	no	0.9608	0.0101	-	-
	≈ 8	≈ 8	no	0.9537	0.0121	-	-
	0	16	no	0.9172	0.0283	-	-

Table 5.2: Segmentation performance comparison between training on real data, generated data, and mixed data, using either no additional data augmentation, or standard data augmentation (see Table 5.1), evaluated on our test set of the SCR Lung Database. Since our previous setup [45] was not tested on full resolution, the Hausdorff distance was omitted from the previous results, as it is not an accurate comparison.

5.1.4 Discussion

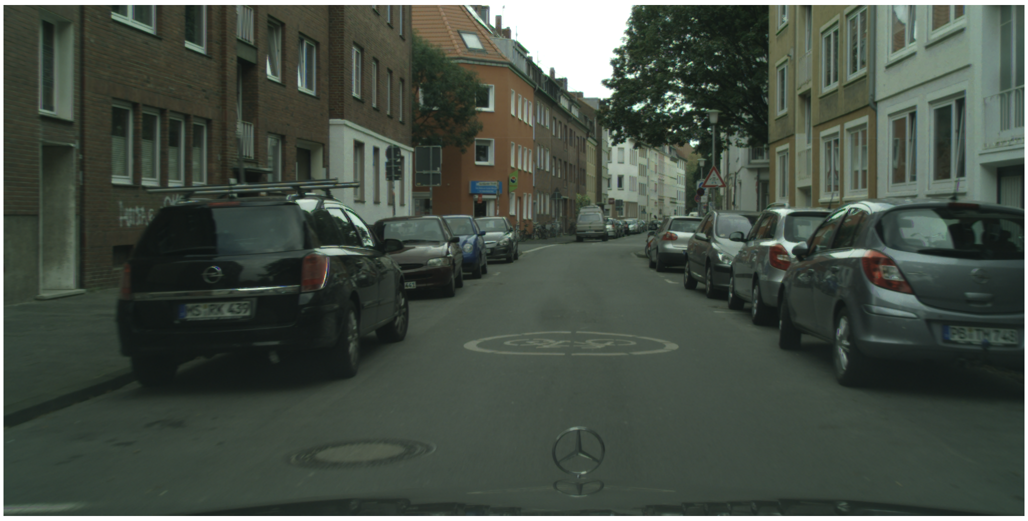
Figure 5.3 shows that our *GAN* manages to generate high-quality images with corresponding segmentation masks that fit the generated image well. Compared to our previous *GAN* samples of this dataset shown in [45], the generated samples are of much higher quality and more closely resemble the training data. We also do not experience mode collapse of our generated samples compared to our previous results, as the resulting samples show similar variety to the training set the generator was trained on. Looking at the segmentation performance shown in Table 5.2, we can see that the Dice scores and Hausdorff distances are very close between all networks, only showing a significant gap for the networks trained on strictly synthetic data and for our previous results in [45]. Augmenting with synthetic images from a trained *GAN* does not decrease the segmentation performance, and networks trained with a mix of synthetic and real images stay competitive with networks trained on strictly real data, using standard data augmentation. Even though the difference is small, the best result (Dice score, standard deviation of Dice) of our evaluation was achieved using our *GAN*-based augmentation, i.e. using a network trained on mixed real and synthetic data. This suggests that *GAN*-based augmentation might be a viable augmentation strategy in the future, especially if *GAN* research further improves on the quality and variety of generated images. Furthermore, it is very interesting to see that our network trained on purely synthetic data beats the performance of the network trained on real data of our previous work [45]. Important to note is that this is not due to mode collapse or overfitting of our generator on the training data, but due to the higher quality of the synthetic images sampled from our *GAN* on-the-fly, in addition to the use of a more powerful segmentation network. This shows that our *GAN* has managed to learn enough about the underlying training data distribution to produce valuable images for training segmentation networks.

Looking at the samples produced from our *GAN*-augmented network and our network trained with standard augmentation in Figure 5.4 and Figure 5.5, we can see that the segmentation quality is also equally good. For some test images, the network trained with *GAN*-based data augmentation produces better segmentation masks, while for others, the network trained with standard data augmentation achieves higher quality results. Since the Dice scores and Hausdorff distances are almost identical, and we can not determine significant differences in image quality, it seems that the lung segmentation problem for this dataset is already solved by the U-Net. Additional augmentation does not provide any more benefits, but also does not have a negative impact on the results either. However, *GAN*-based augmentation also does not lead to worse performance in this case, which was not the case in our previous evaluation [45], suggesting that the higher quality *GAN* images from Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) improved our overall augmentation method, and our *GAN* managed to better capture the distribution of our training data.

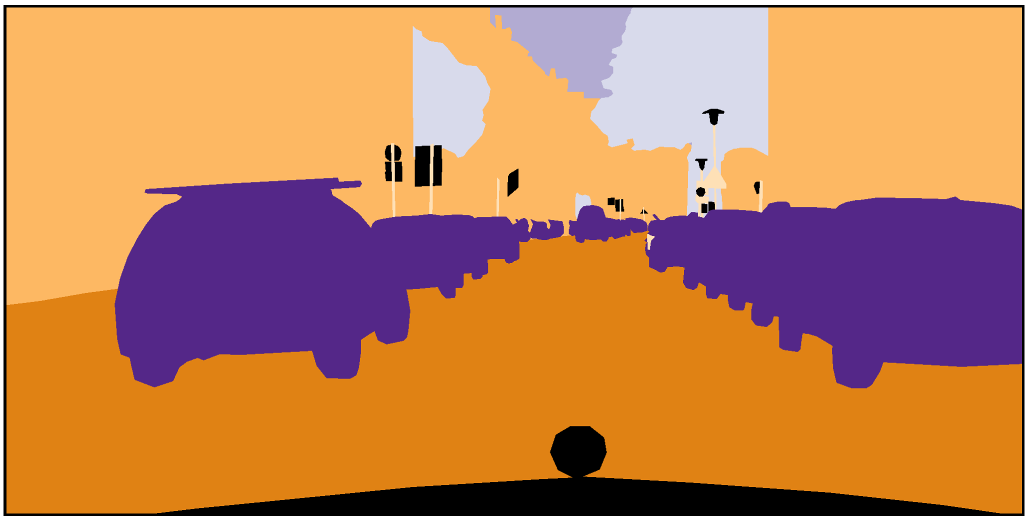
5.2 Cityscapes Semantic Image Segmentation

5.2.1 Dataset Description

For our second evaluation, we chose the task of semantic segmentation using the *Cityscapes* [10] dataset. Cityscapes is a challenging dataset for semantic urban scene understanding, which aims to capture the complexity of real-world urban scenes. For 30 object classes divided into 8 groups, pixel-level and instance-level segmentation masks are provided for every image. The base resolution of all images is $[2048 \times 1024 \times 3]$. This dataset consists of 2975 training images and 500 validation images with finely annotated segmentation masks, with an online submission system used to evaluate performance on the test set, for which the groundtruth segmentation masks are not known. Since this segmentation problem is much more challenging compared to the lung segmentation problem we evaluated in Section 5.1, we decided to only do segmentation of the 8 object groups (*'categories'*) defined in the Cityscapes dataset, and not on the individual classes. An example image as well as the corresponding segmentation mask containing these 8 categories can be seen in Figure 5.6.



(a) Example image from the Cityscapes dataset.



(b) Example segmentation mask from the Cityscapes dataset. Each of the 8 categories and its color-coding are listed in the legend below.

Categories:

⬛	Void	🟧	Flat
🟧	Construction	🟨	Object
⬜	Nature	🟪	Sky
🟪	Human	🟪	Vehicle

Figure 5.6: Example image and category segmentation mask from the Cityscapes dataset.

5.2.2 Evaluation Procedure

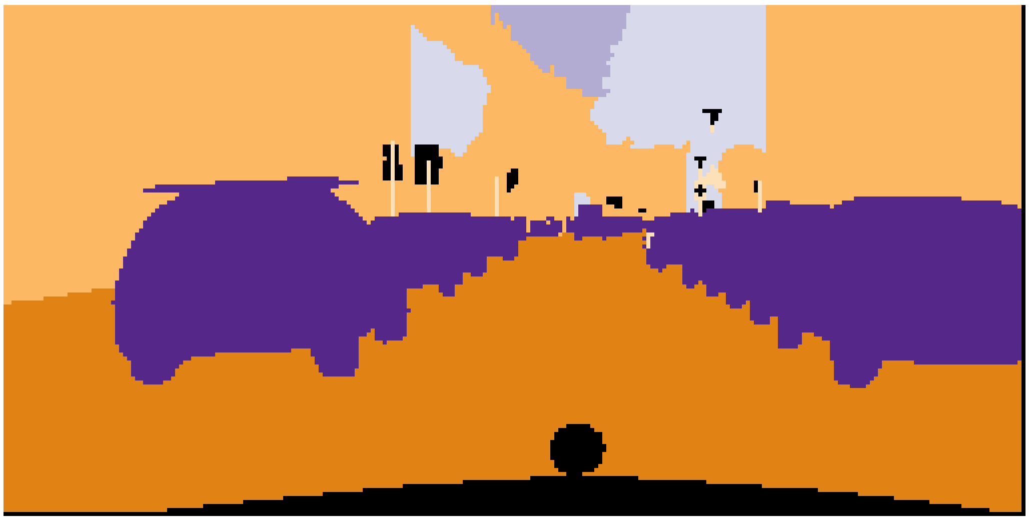
Due to the test set being not as easily accessible, we created our own data split from the given training and validation sets. We used all 500 images from the Cityscapes validation set as our test set. For our internal validation set, we randomly selected 400 images from the Cityscapes training set. Finally, our training set consists of the remaining 2575 images from the Cityscapes training set. Due to the much larger amount of data compared to the SCR Lung Database and the time consuming nature of our evaluation, we only evaluate on this single fold of data. For all networks in this evaluation, we downscaled the resolution of all input images to $[256 \times 128 \times 3]$ to be able to fit our generator network and our segmentation network into memory at the same time, while still keeping a sufficiently large minibatch size. This is definitely not ideal, as the Cityscapes dataset contains a lot of small, thin structures (e.g. objects such as street lights and traffic signs), that almost vanish in the downsampled image. An example for such a downsampled input is shown in Figure 5.7.

Before training our segmentation networks, we train our modified *GAN* (see Section 4.1) on the Cityscapes dataset for 10000 iterations, as the image quality did not improve further after that. For the standard data augmentation, we experimented using a set of multiple different augmentation methods, and chose the best one based on the performance on the validation set. The validation results for different augmentation methods are shown in Table 5.3, and the combination of parameters listed in bold will be used as our standard data augmentation method for further training. Similar to before, we threshold the output segmentation images of the generator to arrive at discrete segmentation masks. An example for a segmentation mask sampled from the *GAN* before and after thresholding can be seen in Figure 5.8.

Augmentation Parameters				Validation Performance
Intensity shift around zero (stddev)	Intensity scaling around zero (stddev)	Random translation around zero (stddev)	Horizontal flipping	*mIoU*
-	**-**	**-**	**yes**	**0.8042**
0.05	0.05	-	yes	0.7950
-	-	-	no	0.7804
0.05	0.05	5.00 px	no	0.7421
0.10	0.10	10.00 px	no	0.6989

Table 5.3: Comparison of augmentation parameters for the Cityscapes dataset. All results were computed by training strictly on real data and recording the best validation performance. For illustrative purposes, we only report the *mIoU*, although the Hausdorff distance and Dice score follow the same ordering of segmentation performance. For further evaluation, we use the augmentation parameters listed in **bold** as our standard data augmentation.

(a) Example of a downsampled image from the Cityscapes dataset.

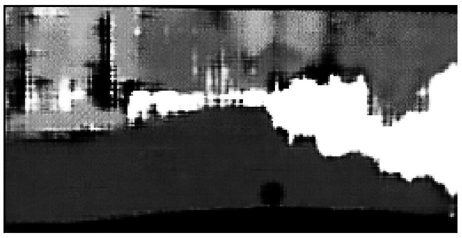


(b) Example of a downsampled segmentation mask from the Cityscapes dataset. Each of the 8 categories and its color-coding are listed in the legend below.

Categories:

⬛ Void	🟧 Flat
🟧 Construction	🟨 Object
⬜ Nature	🟪 Sky
🟪 Human	🟪 Vehicle

Figure 5.7: Example of a downsampled image and category segmentation mask from the Cityscapes dataset.


(a) Raw segmentation image sampled from the *GAN* in the intensity range of $[-1, 1]$.

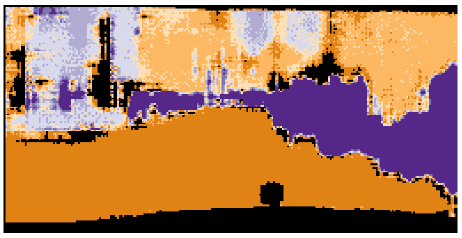

(b) Discrete segmentation mask after thresholding the *GAN* output.

Figure 5.8: Raw output and thresholded segmentation mask sampled from the *GAN* trained on the Cityscapes dataset. Segmentation classes in the thresholded image are the same as in Figure 5.6, and use the same color mapping for visualization.

For our final evaluation of the Cityscapes dataset, we again train our segmentation network on different ratios of real and generated data, using either no augmentation or standard data augmentation. Each segmentation network took approximately 24 hours to train until convergence.

5.2.3 Results

The full details of our implementation and network architectures are described in Section 4.2. Samples of our *GAN* trained on Cityscapes are shown in Figure 5.10. Our final segmentation performance for all different evaluation setups of the Cityscapes dataset is documented in Table 5.4. Additionally, we show the resulting *mIoU* for every category for the four best performing networks in Figure 5.9. Since the significant amount of downsampling of the input images makes the segmentation of small objects of the categories *'Human'* and *'Object'* much more difficult, we also present *mIoU* results excluding those categories in Table 5.4. Similar to our evaluation in Section 5.1, we also show several examples of resulting segmentation masks to better compare *GAN*-based augmentation to standard augmentation. These examples are illustrated in Figure 5.11. In order to better demonstrate the variety of the Cityscapes dataset, we show additional examples of good segmentation masks in Figure 5.12.

5.2.4 Discussion

By looking at the sample images shown in Figure 5.10, we can see that the image quality is not as high when compared to the *GAN* samples shown in Section 5.1.3. Still, the network learned to generate images of good variety, without the problem of mode collapse (see Section 3.1). Most importantly, the generated segmentation masks still fit to every generated image. *GANs* have consistently shown to generate very realistic images, especially for datasets containing a large amount of self-similarity, such as face datasets [31]. In addition to self-similarity, the most impressive *GAN* results are often achieved by training on

Network ID	Number of real pairs in minibatch	Number of generated pairs in minibatch	Aug.?	*mIoU*	*mIoU* excluding 'Human' and 'Object'
8-0-Aug	8	0	yes	**0.7859**	**0.8894**
8-0-NoAug	8	0	no	0.7616	0.8767
4-4-Aug	4	4	yes	0.7548	0.8732
4-4-NoAug	4	4	no	0.7630	0.8786
0-8-Aug	0	8	yes	0.4705	0.6535
0-8-NoAug	0	8	no	0.4632	0.6426

Table 5.4: Segmentation performance comparison between training on real data, generated data, and mixed data, using either no additional data augmentation, or standard data augmentation, evaluated on our test set of the Cityscapes dataset.

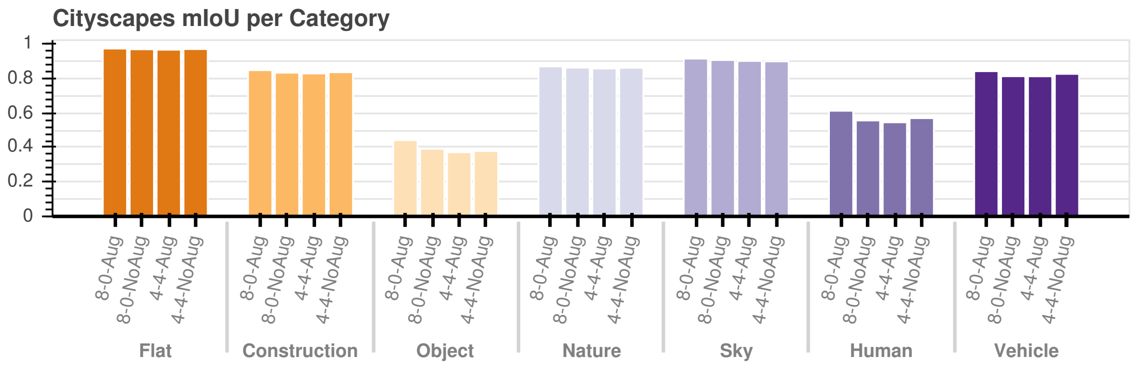


Figure 5.9: *mIoU* for every category of the Cityscapes dataset. For visual clarity, we omit the worst performing networks and only report results for the four best networks, identified by their network ID shown in Table 5.4.

huge datasets, containing millions of images. However, the usage of *GANs* as a generative model for more complex datasets has not been explored in much detail in related literature so far, therefore it is very interesting to see how the resulting images of an off-the-shelf *GAN* for a complex dataset look like. Especially comparing to Figure 5.10, we can see that self-similarity seems to significantly improve the image quality, and varying datasets such as Cityscapes seem to be very difficult to learn for a *GAN*.

For our quantitative evaluation, we found that the best standard data augmentation for this dataset and our segmentation network architecture was to just use horizontal flipping (see Table 5.3). Using other combinations of intensity shift, intensity scaling, or random translation led to worse segmentation performance. For some settings, the segmentation performance was even worse than not using data augmentation at all. This illustrates an important point of data augmentation - the augmentation parameters require careful tuning to fit the dataset, as wrong data augmentation can have a negative effect by dras-

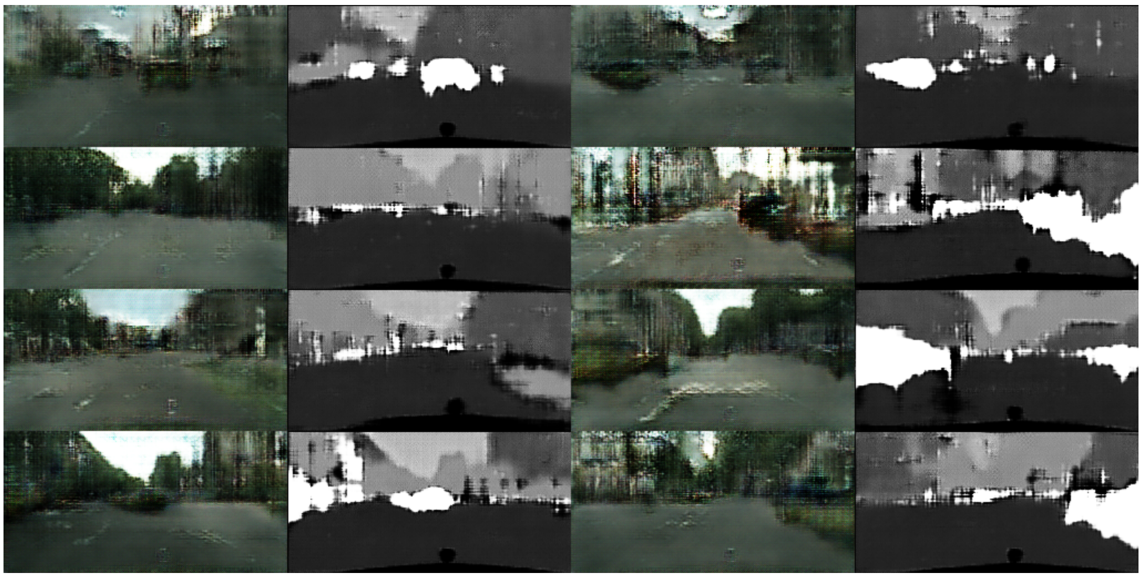


Figure 5.10: Sample images from our Generative Adversarial Network trained on the Cityscapes dataset. Columns 1 and 3 show generated images, while columns 2 and 4 show the respective generated segmentation masks.

tically reducing the segmentation performance. From our final segmentation performance shown in Table 5.4, we can see that the network trained on real data, using standard data augmentation, achieved the best performance compared to all other networks. However, we can again observe that the network trained using *GAN*-based augmentation without additional standard data augmentation achieves similar performance to the network that was trained on real data without augmentation. Especially interesting is that when using *GAN*-based augmentation without standard augmentation, the results are actually better than some of the results when using standard augmentation shown in Table 5.3. This illustrates that our *GAN* has learned a reasonable representation of our training data, even though the generated samples are not of high quality.

Compared to the highscore database of the Cityscapes dataset[1], our baseline performance for the category *mIoU* is in line with the weaker results on the online database. Our intuition is that the reason for this is twofold. First, we perform 8-times downsampling of the training images in order to fit our computational budget. On one hand, this is because it is very difficult to train *GANs* on even higher resolutions, as for higher resolution images, discriminating between generated and real images is easier [49], resulting in gradients that point in random directions, which are thereful not useful for learning [2]. Additionally, high-resolution deep networks typically require a lower minibatch size to fit into GPU memory, therefore further compromising training stability, unless very recent,

[1]Cityscapes Pixel-Level Semantic Labeling Task Results, `https://www.cityscapes-dataset.com/benchmarks/#pixel-level-results`, Accessed: 20.02.2018

more sophisticated methods, such as ProGAN [31], are used. On the other hand, due to the nature of our evaluation setup shown in Figure 4.2, we require the full generator model and our segmentation network to both fit into memory, as we sample the generator during training of our segmentation network. Therefore, we quickly reach our GPU memory limit, forcing us to downsample the input data.

Performing large amounts of downsampling leads to a much worse segmentation performance for small or thin structures, and borders between regions, as those fine details vanish when downsampling is applied. When comparing the example image shown in Figure 5.6 to its downsampled version shown in Figure 5.7, it is easy to see that fine details are almost completely lost. The traffic sign posts that are easily visible in the full-resolution image are only one pixel wide on the downsampled image, making segmentation of those regions very difficult. This effect can also be seen by comparing the resulting segmentation masks of our networks, shown in Figure 5.11 and Figure 5.12. Most of the errors of our results are in the border regions between classes, as the fine detail necessary to determine exact borders is lost during downsampling. We also observe the consequence of downsampling in Figure 5.9, where we show results for every category. While our networks consistently perform much worse on the 'Object' and 'Human' categories, the other categories show good results, given that we only used a standard U-Net segmentation network architecture with additional data augmentation methods. Computing the *mIoU* over all categories but those two, we achieve much better scores, as can be seen in Table 5.4.

The second reason for our weaker performance compared to the competition on the highscore list is that we do not pretrain our networks. Many of the submissions on the highscore list use pre-trained network weights, mostly trained on the ImageNet [59] database, as feature extractors in their network pipeline. As one of our main goals is to evaluate how *GAN*-based data augmentation affects the results of training segmentation networks, we did not want to additionally pre-train our networks, as that introduces another variable that significantly impacts training behavior of deep networks.

The effects of downsampling can also be seen in the example segmentation masks produced by our networks shown in Figure 5.11. Both the network trained using *GAN*-based augmentation and the network trained using standard data augmentation produce reasonable results. However, they both suffer from similar problems. While large regions are detected well, small objects are often completely missing, and borders between class regions are not as precise. This is especially visible in the error images, which show that most of the error comes from imprecise borders and small objects. However, we can see that for easier images our networks produce high-quality segmentation masks, and even our worst results shown in Figure 5.11 still look reasonably good.

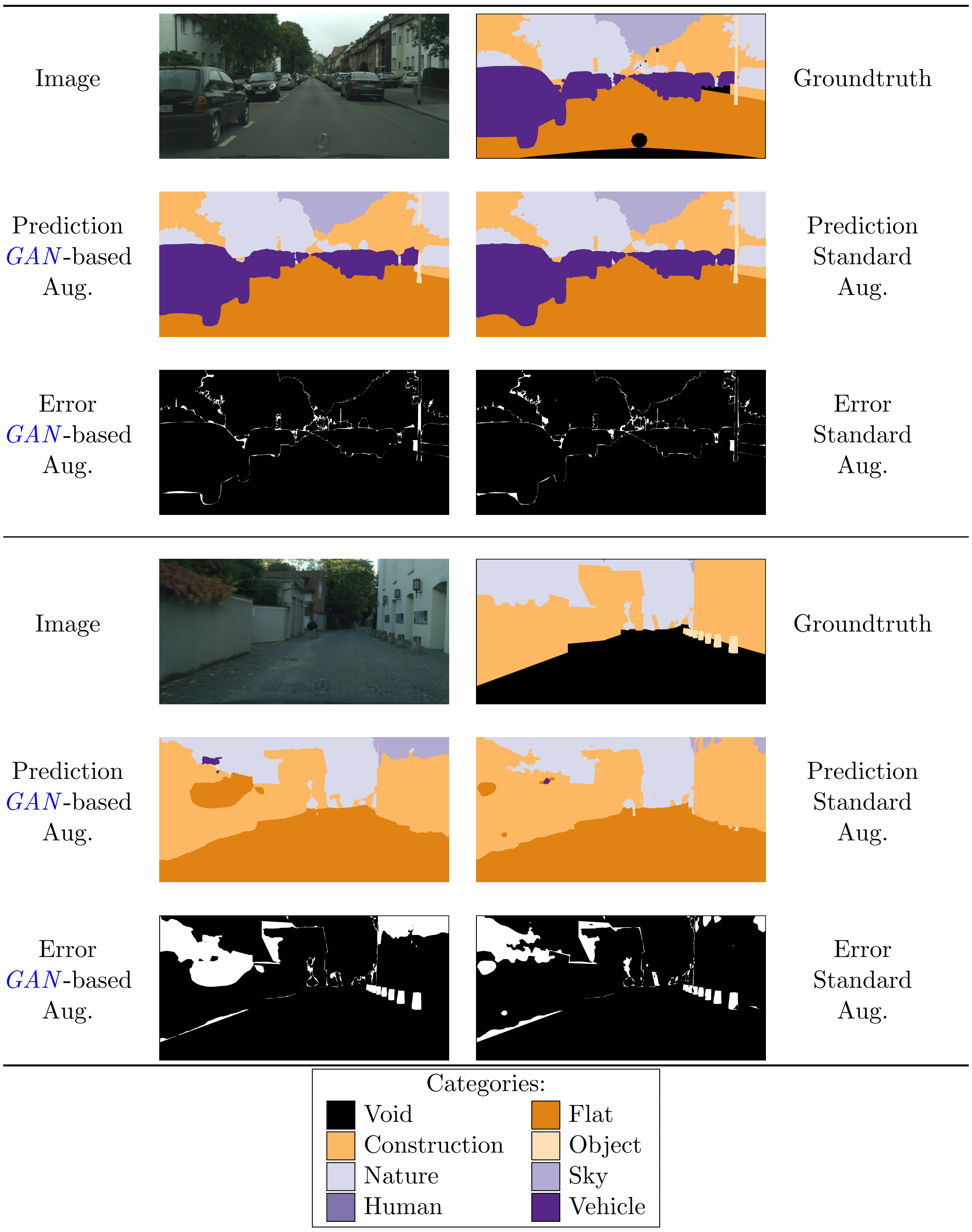


Figure 5.11: Comparison of segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the Cityscapes dataset. The comparison on the top shows the best test result, while the comparison on the bottom shows the worst result.

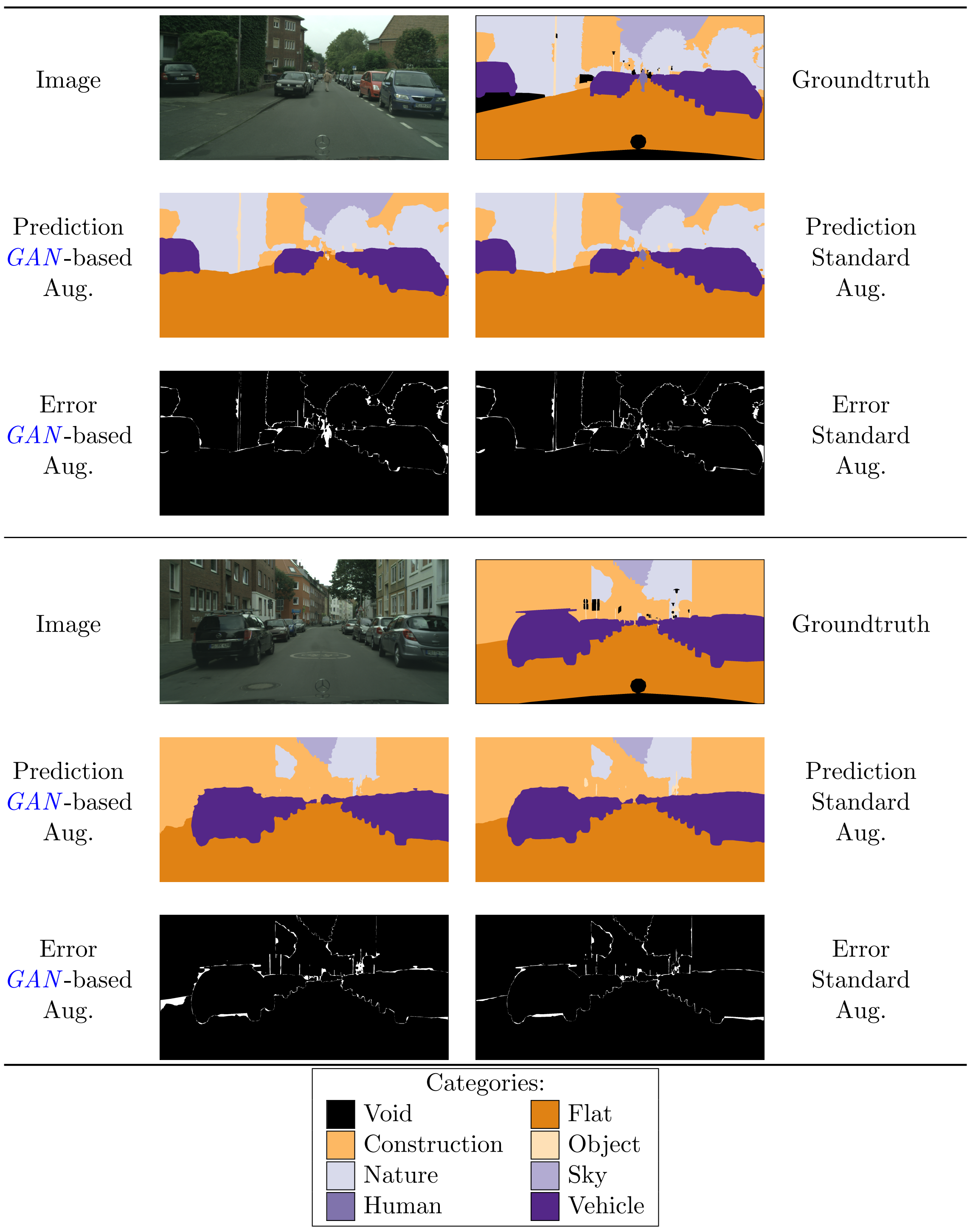


Figure 5.12: Comparison of segmentation masks from fully trained segmentation networks between standard data augmentation and *GAN*-based data augmentation for the Cityscapes dataset. Both results achieved good segmentation performance.

6 Conclusion

Data augmentation in deep learning presents an easy to use method of regularization, especially when training data is scarce. Due to the sheer amount of possible augmentation methods, it is necessary to carefully analyze the training data and find a fitting augmentation method to improve the performance of deep networks. While data augmentation is typically done using simple image transformations, the recent popularity of Generative Adversarial Networks (GANs) led to new possibilties for data augmentation. Even though previous research was done on data augmentation using synthetic data, for example by rendering photo-realistic images, *GANs* illustrate an interesting new approach in learning a generative model from data. *GANs* provide the benefit of learning a generative model while being able to use conventional deep learning techniques. While it was difficult to train *GANs* when they were first introduced in 2014, especially with recent advances such as the Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP), training *GANs* has become easier and more stable.

During the course of this thesis, we proposed a novel *GAN* architecture [45], which is able to generate image-segmentation pairs, instead of just images. This modification allows us to use the generated data as training data for supervised segmentation networks, therefore augmenting the original training data with synthetic data, generated by the *GAN*. Although our initial published results were promising [45], the sample quality of our *GAN* and our evaluation setup showed a lot of possibilities for improvement, especially as we did not compare standard data augmentation to our *GAN*-based approach.

Our main focus of this thesis was to perform a comparison between standard data augmentation and *GAN*-based data augmentation. For the first comparison, we chose to perform medical image segmentation of the SCR Lung Database, similar to our previously published results. We found that by making use of more recent advancements in *GAN* research, such as *WGAN-GP*, we can increase the image resolution even further, compared to our previous approach, leading to higher image quality. However, as we also improved our segmentation network architecture, we seemed to already approach saturation in segmentation performance. Both *GAN*-based augmentation and standard data augmentation produced almost identical results compared to using no augmentation at all. Still, *GAN*-based augmentation achieved competitive performance compared to standard data augmentation, and networks trained on purely synthetic data still achieved good results, further illustrating that our *GAN* has managed to capture the training data well enough to be used for training other networks.

Our second comparison focused on a much more challenging dataset, the Cityscapes dataset for urban scene understanding. Most research work done on training *GANs* uses huge datasets with a lot of self-similarity, such as human face datasets. One of our core motivations to choose a more difficult dataset was to test how well our *GAN* architecture handles the generation of complex, multi-class segmentation masks. We found that our *GAN* learned to generate images resembling our training data, although the image quality was much lower compared to the synthetically generated images based on the SCR Lung Database. Our intuition is that this is mostly because of the missing self-similarity and the relatively low amount of images, given the complexity of the dataset. For this evaluation, we observe that choosing a correct data augmentation setup has significant impact on the segmentation performance. We found that for some augmentation methods, the segmentation performance decreased, even past our baseline non-augmented segmentation network. Due to the lower image quality of the generated images, the segmentation performance did not improve when using *GAN*-based augmentation. However, similar to our evaluation done on the SCR Lung Database, using additional *GAN* samples for training did not decrease the segmentation performance compared to using no augmentation. As this dataset is much more challenging, we experienced a significant increase in performance when using standard data augmentation.

To summarize, we performed an extensive evaluation of the possibilities of using *GANs* for training data augmentation in image segmentation tasks. While our current results only show that *GAN*-based augmentation neither has a positive, nor a negative impact, we believe that if a *GAN* was able to fully learn the training data distribution, the additional synthetic data could be highly useful as a regularizer for deep networks. Compared to standard data augmentation, *GAN*-based augmentation does not require extensive data analysis to find out optimal augmentation parameters. Especially in the Cityscapes evaluation, we saw how certain data augmentation parameters can lead to much worse performance, therefore an augmentation method that is learned from data would save a lot of effort in fine-tuning deep networks. Future research could focus on using our modified *GAN* architecture in different setups. Popular image-to-image translation *GANs* such as CycleGAN [78] could be modified to generate image-segmentation pairs, instead of just images. This could then be used to translate a whole annotated dataset into another domain, where it can be used as synthetic data for training data augmentation. Additionally, the most obvious future improvement of our work would be to increase the resolution and representation power of our *GAN*, leading to higher quality synthetic images. We are certain that such an improved generative model could be used as a data augmentation method to improve performance for supervised deep learning tasks.

A List of Acronyms

Adam	Adaptive Moment Estimation 15, 16, 52, 61–64
CNN	Convolutional Neural Network 6, 8, 25–27, 30–32, 38, 41, 47, 48
DCGAN	Deep Convolutional Generative Adversarial Network xvi, 43, 47–49, 52, 56, 60, 62
EM	Earth-Mover 50, 51, 56
FCNN	Fully-Convolutional Neural Network 47
FNN	Feedforward Neural Network xv, 6–13, 15, 17–20, 24, 32, 41
GAN	Generative Adversarial Network iii, v, vi, xvi, xvii, 3, 4, 30, 41, 43–51, 53–63, 68–73, 75, 77–84
IoU	Intersection-over-Union 65
JS	Jensen-Shannon 49, 50
KL	Kullback-Leibler 49, 50
leaky ReLU	Leaky Rectified Linear Unit xv, 24, 47
mIoU	mean Intersection-over-Union xvi, 60, 70, 75, 77–80
MSE	Mean Squared Error 9, 10, 14
ReLU	Rectified Linear Unit xv, 13, 17, 23, 24, 45, 47, 61–64
ResNet	Residual Network 13
SGD	Stochastic Gradient Descent 6, 13, 15, 16, 19, 23, 25, 35, 41, 45, 56
tanh	Hyperbolic Tangent xv, 21–23, 47, 61, 63

WGAN	Wasserstein Generative Adversarial Network 43, 49–52, 56, 58
WGAN-GP	Wasserstein Generative Adversarial Network with Gradient Penalty xvi, 4, 43, 51–53, 56, 58–60, 73, 83

B List of Publications

My work at the Institute of Computer Graphics and Vision led to the following peer-reviewed publications.

B.1 2017

Generative Adversarial Network based Synthesis for Supervised Medical Image Segmentation

Thomas Neff, Christian Payer, Darko Štern and Martin Urschler
In: *Proceedings of the OAGM&ARW Joint Workshop*
May 2017, Vienna, Austria
pages 140-145.
(Accepted for oral presentation, **Best Paper Award**)

Abstract: Modern deep learning methods achieve state-of-the-art results in many computer vision tasks. While these methods perform well when trained on large datasets, deep learning methods suffer from overfitting and lack of generalization given smaller datasets. Especially in medical image analysis, acquisition of both imaging data and corresponding ground-truth annotations (e.g. pixel-wise segmentation masks) as required for supervised tasks, is time consuming and costly, since experts are needed to manually annotate data. In this work we study this problem by proposing a new variant of Generative Adversarial Networks (GANs), which, in addition to synthesized medical images, also generates segmentation masks for the use in supervised medical image analysis applications. We evaluate our approach on a lung segmentation task involving thorax X-ray images, and show that GANs have the potential to be used for synthesizing training data in this specific application.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, OSDI'16, pages 265–283. USENIX Association. (page 59)

[2] Arjovsky, M. and Bottou, L. (2017). Towards principled methods for training Generative Adversarial Networks. In *Proceedings of the Fifth International Conference on Learning Representations (ICLR)*. (page 50, 79)

[3] Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein Generative Adversarial Networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning (ICML)*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223. PMLR. (page 43, 47, 49, 50)

[4] Ba, L. J., Kiros, R., and Hinton, G. E. (2016). Layer normalization. In *NIPS 2016 Deep Learning Symposium*. (page 52)

[5] Badrinarayanan, V., Kendall, A., and Cipolla, R. (2017). SegNet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 39(12):2481–2495. (page 30)

[6] Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 437–478. Springer. (page 17)

[7] Berthelot, D., Schumm, T., and Metz, L. (2017). BEGAN: Boundary equilibrium generative adversarial networks. *CoRR*, abs/1703.10717. (page 43)

[8] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Lechevallier, Y. and Saporta, G., editors, *Proceedings of COMPSTAT2010: 19th International Conference on Computational Statistics*, pages 177–186. Physica-Verlag HD, Heidelberg. (page 6)

[9] Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L. (2015). Semantic image segmentation with deep convolutional nets and fully connected CRFs. In *Proceedings of the Third International Conference of Learning Representations (ICLR)*. (page 28)

[10] Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., and Schiele, B. (2016). The cityscapes dataset for semantic urban scene

understanding. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (page 4, 57, 73)

[11] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314. (page 8)

[12] Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302. (page 64)

[13] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*, 12:2121–2159. (page 15)

[14] Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2012). Scene parsing with multiscale feature learning, purity trees, and optimal covers. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML'12, pages 1857–1864. Omnipress. (page 1)

[15] Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., Villena-Martinez, V., and Rodríguez, J. G. (2017). A review on deep learning techniques applied to semantic segmentation. *CoRR*, abs/1704.06857. (page 32)

[16] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9, pages 249–256. (page 17)

[17] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323. PMLR. (page 45, 47)

[18] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`. (page 1, 2, 7, 8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23, 25, 27)

[19] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27 (NIPS)*, pages 2672–2680. (page 3, 4, 43, 44, 45, 46, 49)

[20] Goodfellow, I. J. (2017). NIPS 2016 tutorial: Generative Adversarial Networks. *CoRR*, abs/1701.00160. (page 44, 46)

[21] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. (2017). Improved training of Wasserstein GANs. In Guyon, I., Luxburg, U. V., Bengio, S.,

Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30 (NIPS)*, pages 5769–5779. Curran Associates, Inc. (page 3, 4, 43, 47, 51, 52, 53, 59)

[22] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034. IEEE Computer Society. (page 17, 24, 61, 62, 63, 64)

[23] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE Computer Society. (page 13, 60)

[24] Hinton, G., Deng, L., Yu, D., Dahl, G. E., r. Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012a). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine (SPM)*, 29(6):82–97. (page 1, 5)

[25] Hinton, G., Srivastava, N., and Swersky, K. (2012b). Lecture 6a: Overview of mini-batch gradient descent. COURSERA: Neural Networks for Machine Learning. (page 15, 16, 51)

[26] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München. (page 12, 23)

[27] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366. (page 8)

[28] Huttenlocher, D. P., Klanderman, G. A., and Rucklidge, W. A. (1993). Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863. (page 65)

[29] Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456. PMLR. (page 13, 20, 47)

[30] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2017). Image-to-image translation with Conditional Adversarial Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (page 3, 43)

[31] Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). Progressive growing of GANs for improved quality, stability, and variation. *Proceedings of the Sixth International Conference on Learning Representations (ICLR)*. (page 3, 43, 77, 80)

[32] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of the Third International Conference on Learning Representations (ICLR)*. (page 15, 16, 52)

[33] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto. (page 46, 48)

[34] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25 (NIPS)*, pages 1097–1105. Curran Associates, Inc. (page 17, 23, 35)

[35] Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems 4 (NIPS)*, pages 950–957. Morgan-Kaufmann. (page 6)

[36] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–444. (page 1, 5)

[37] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551. (page 25)

[38] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. (page 46)

[39] Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., and Shi, W. (2017). Photo-realistic single image super-resolution using a Generative Adversarial Network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114. (page 43)

[40] Li, F.-F., Johnson, J., and Yeung, S. (2017). CS231n: Convolutional neural networks for visual recognition 2017. (page 15, 16, 17, 21, 23, 24, 25, 27, 30)

[41] Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL)*. (page 47)

[42] Mao, X., Li, Q., Xie, H., Lau, R. Y. K., and Wang, Z. (2016). Least-squares Generative Adversarial Networks. *CoRR*, abs/1611.04076. (page 43)

[43] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133. (page 6)

[44] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814. Omnipress. (page 13, 23)

[45] Neff, T., Payer, C., Štern, D., and Urschler, M. (2017). Generative Adversarial Network based synthesis for supervised medical image segmentation. In *Proceedings of the OAGM&ARW Joint Workshop*, pages 140–145. (page 3, 43, 57, 59, 72, 73, 83)

[46] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. (page 48)

[47] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. (page 6, 11, 12, 13, 34)

[48] Nowozin, S., Cseke, B., and Tomioka, R. (2016). f-GAN: Training generative neural samplers using variational divergence minimization. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29 (NIPS)*, pages 271–279. Curran Associates, Inc. (page 43, 49)

[49] Odena, A., Olah, C., and Shlens, J. (2017). Conditional image synthesis with auxiliary classifier GANs. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651. PMLR. (page 79)

[50] Payer, C., Štern, D., Bischof, H., and Urschler, M. (2016). Regressing heatmaps for multiple landmark localization using CNNs. In Ourselin, S., Joskowicz, L., Sabuncu, M. R., Unal, G., and Wells, W., editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016*, pages 230–238, Cham. Springer. (page 6)

[51] Pham, D. L., Xu, C., and Prince, J. L. (2000). Current methods in medical image segmentation. *Annual Review of Biomedical Engineering*, 2(1):315–337. (page 6)

[52] Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with Deep Convolutional Generative Adversarial Networks. In *Proceedings of the Fourth International Conference of Learning Representations (ICLR)*. (page 30, 43, 47, 48, 49, 60, 62)

[53] Riegler, G., Urschler, M., Rüther, M., Bischof, H., and Štern, D. (2015). Anatomical landmark detection in medical applications driven by synthetic data. In *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, pages 85–89. (page 3, 41)

[54] Ronneberger, O., Fischer, P., and Brox, T. (2015). U-Net: Convolutional networks for biomedical image segmentation. In Navab, N., Hornegger, J., Wells, W. M., and Frangi, A. F., editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham. Springer. (page 1, 5, 30, 33, 35, 40, 59, 60, 62)

[55] Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408. (page 7)

[56] Rozantsev, A., Lepetit, V., and Fua, P. (2015). On rendering synthetic images for training an object detector. *Computer Vision and Image Understanding (CVIU)*, 137:24 – 37. (page 3, 41)

[57] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747. (page 15, 16)

[58] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536. (page 7, 10, 11)

[59] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252. (page 35, 80)

[60] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X., and Chen, X. (2016). Improved techniques for training GANs. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29 (NIPS)*, pages 2234–2242. Curran Associates, Inc. (page 46, 47)

[61] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117. (page 11)

[62] Shelhamer, E., Long, J., and Darrell, T. (2017). Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 39(4):640–651. (page 6)

[63] Shiraishi, J., Katsuragawa, S., Ikezoe, J., Matsumoto, T., Kobayashi, T., Komatsu, K.-i., Matsui, M., Fujita, H., Kodera, Y., and Doi, K. (2000). Development of a digital image database for chest radiographs with and without a lung nodule. *American Journal of Roentgenology (AJR)*, 174(1):71–74. (page 67)

[64] Shrivastava, A., Pfister, T., Tuzel, O., Susskind, J., Wang, W., and Webb, R. (2017). Learning from simulated and unsupervised images through adversarial training. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2242–2251. (page 3, 4, 41, 43, 53, 54, 55)

[65] Sørensen, T. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biologiske Skrifter*, 5:1–34. (page 64)

[66] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806. (page 30)

[67] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958. (page 6, 19, 45)

[68] Stockman, G. and Shapiro, L. G. (2001). *Computer Vision*. Prentice Hall PTR, 1st edition. (page 3, 31, 32)

[69] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, volume 28, pages III–1139–III–1147. JMLR.org. (page 14)

[70] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*, volume 2, pages 3104–3112. MIT Press. (page 1, 5)

[71] Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). DeepFace: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1701–1708. (page 1, 5)

[72] van Ginneken, B., Stegmann, M., and Loog, M. (2006). Segmentation of anatomical structures in chest radiographs using supervised methods: a comparative study on a public database. *Medical Image Analysis*, 10(1):19–40. (page 4, 57, 67)

[73] Villani, C. (2008). *Optimal Transport*. Springer. (page 50)

[74] Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference*, pages 762–770. (page 11)

[75] Yu, F. and Koltun, V. (2016). Multi-scale context aggregation by dilated convolutions. In *Proceedings of the Fourth International Conference of Learning Representations (ICLR)*. (page 28)

[76] Yu, F., Zhang, Y., Song, S., Seff, A., and Xiao, J. (2015). LSUN: construction of a large-scale image dataset using deep learning with humans in the loop. *CoRR*, abs/1506.03365. (page 47, 48, 49, 52, 53)

[77] Zhao, J. J., Mathieu, M., and LeCun, Y. (2017). Energy-based Generative Adversarial Networks. In *Proceedings of the Fifth International Conference of Learning Representations (ICLR)*. (page 43, 49)

[78] Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. (2017). Unpaired image-to-image translation using Cycle-Consistent Adversarial Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*. (page 3, 43, 84)