David Jakob Schinagl, BSc

Generative Adversarial Networks Properties & Applications

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Telematics

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Horst Bischof
Institute for Computer Graphics and Vision

Advisors

Dipl.-Ing. Gernot Riegler
Dipl.-Ing. Dr.techn. Samuel Schulter
Institute for Computer Graphics and Vision

Graz, Austria, May. 2016

Abstract

In recent years, deep learning methods gained much attention in the field of computer vision. They achieve outstanding results in discriminative tasks like image classification, where a high-dimensional input is mapped to a class label. In contrast, deep generative models did not reach this level of success until recently. Generative models capture the underlying generation process of the data and can be used to synthesize new samples. A new approach based on artificial neural networks, called generative adversarial networks (GANs), represents an attractive alternative to existing generative models based on maximum likelihood estimation and performs well on various datasets. However, the internal generation process of GANs, from the initial noise vector to the resulting image is relatively unexplored.

In this work we investigate the internals of adversarial nets more deeply and demonstrate the universal usability of this model based on two applications. In the first part, GANs are trained on depth-datasets and the resulting networks are analyzed in a variety of ways. We explore the latent noise space to investigate how semantic properties of the synthesized samples are encoded within this space. Moreover, we present two methods to influence the generation process in order to synthesize depth-data with desired properties. In the second part, GANs are applied to two fundamental computer vision tasks: The first one is unsupervised feature learning where we demonstrate that the features learned by the adversarial networks are useful for classification and regression tasks when labeled data are scarce. Finally, GANs are applied to domain specific image super resolution where we show that adversarial nets can be used to significantly increase the quality of upsampled face images.

Kurzfassung

In den letzten Jahren haben *Deep Learning* Methoden auf dem Gebiet des maschinellen Sehens immer mehr Aufmerksamkeit auf sich gezogen. Vor allem bei diskriminativen Problemstellungen, wie der Bildklassifikation, haben diese Techniken herausragende Ergebnisse erzielt. Im Gegensatz dazu konnten generative Modelle bisher nicht an diese Erfolge anschließen. Generative Modelle versuchen den zugrunde liegenden Entstehungsprozess zu modellieren und können dann auch dafür verwendet werden neue Daten zu erzeugen. Mit *Generative Adversarial Networks* (GANs) gibt es einen neuen Ansatz basierend auf künstlichen neuronalen Netzen, welcher eine attraktive Alternative zu bestehenden Maximum-Likelihood Modellen darstellt und bereits auf mehrere Datensätze erfolgreich angewandt wurde. Allerdings ist der interne Modellierungsprozess von einem Zufallsvektor zum resultierenden Bild noch relativ unerforscht.

In dieser Arbeit untersuchen wir Eigenschaften von GANs genauer und zeigen deren universelle Einsetzbarkeit anhand zweier Anwendungen. Im ersten Teil der Arbeit trainieren wir GANs auf Tiefendaten und analysieren die resultierenden Netzwerke auf unterschiedliche Weise. Dabei untersuchen wir unter anderem wie Eigenschaften der synthetisierten Bilder in den Zufallsvektoren abgebildet werden. Des Weiteren zeigen wir Wege auf, wie man den internen Modellierungsprozess beeinflussen kann um gewünschte Tiefendaten zu erzeugen. Im zweiten Teil wenden wir GANs auf zwei wichtige Problemstellungen des maschinellen Sehens an: Erstens zeigen wir, dass die von GANs gelernten Datenrepresentationen auch für andere Anwendungen nützlich sein können (*Unsupervised Feature Learning*) und danach demonstrieren wir wie GANs verwendet werden können um die Qualität von stark vergrößerten Bildern zu erhöhen.

Acknowledgments

First, I want to thank Prof. Horst Bischof for being my supervisor and for providing me the opportunity to do this thesis at the Institute for Computer Graphics and Vision (ICG). Additionally, I would like to express my gratitude to my advisors Gernot Riegler and Samuel Schulter who had always an open ear for my questions and pointed me to the right direction. Especially I want to thank Gernot for proof-reading this thesis and all the useful hints over the last months.

At this point, I would also like to thank my study colleagues for their support and friendship. Especially Erich, Georg and Michael, who I shared an office with, for the stimulating thesis related discussions and the fun time we had in between. Moreover I want to thank Stefan who accompanied me since the very first lecture and became a great pal.

Next, I want to thank my family, especially my loving parents who are always there when I need them, wherever I am and whatever I do. Moreover, I would like to show my gratitude to Barbara's family for their support and for letting me part of their family. Above all, I want to express my deepest gratitude to my beloved girlfriend Barbara, who always gives me support & encouragement when I need it most and makes my life better. Finally, I want to dedicate this thesis to my beloved daughter Marlene. Your smile brightens my day, every day.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

Today, machine learning is an active research topic in the field of computer science with many practical applications. We often get in touch with this artificial intelligence technique several times a day without us being aware of it. Whenever we use a web search-engine, smart-phone voice commands or in-camera face detection, we benefit from machine learning. Arthur Samuel (1959) defined it as the field of study that gives computers the ability to learn without being explicitly programmed [47]. Learning in this context, means to recognize patterns, make intelligent decisions and improve with experience.

Especially deep learning methods have recently gained much attention in this field of study. Conventional machine learning approaches are limited in their capability of processing raw data and needed a preprocessing step called feature extraction. This is necessary mainly for two reasons: Firstly, the input data is usually too high-dimensional for direct use. A 128×128 pixels RGB image for instance, consists of $49,152$ variables. In addition to the dimensionality reduction, the second purpose is to transform the data into a representation where the machine learning algorithm is able to solve the problem. For instance, the optimal feature space for classification would be a representation with a minimal variance within a class (intra-class variance), and a

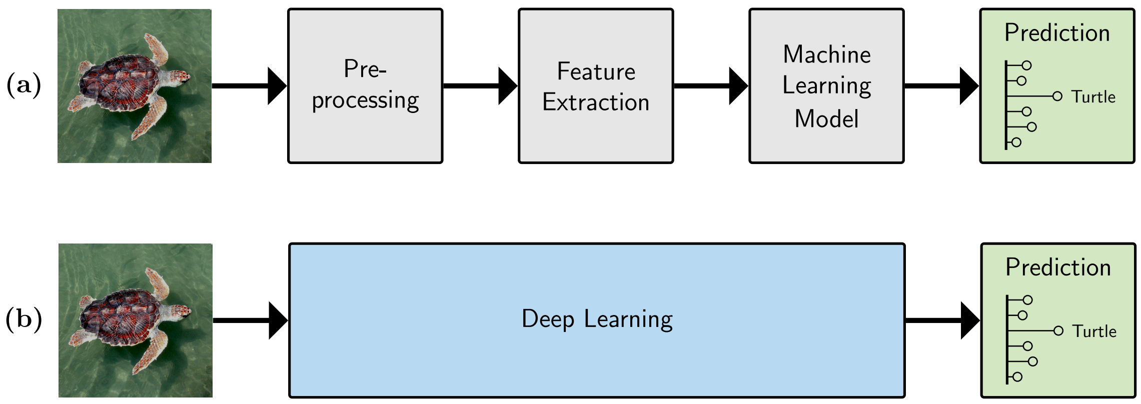


Figure 1.1: Difference between a classic machine learning pipeline and deep learning: (a) A classic machine learning pipeline includes several hand-crafted steps to transform the input into feature space before learning algorithms are used. In contrast to deep learning (b), where this data transformation is also learned.

maximal separation between the classes (inter-class variance). The final performance heavily relied on this features extraction step. However, for complex applications like object detection or handwritten digit recognition it is difficult to know what features should be used. Another problem is, that a useful representation for one dataset is often useless for other data. This problems are addressed by deep learning, where also the data representation is learned. It independently discovers the data representations that are beneficial for the particular task. The difference between these machine learning pipelines is visualized in Figure 1.1. It allows to model hierarchical data representations, where complex high-level features are based on simpler ones. In terms of an image, high-level features like shapes, are formed by lower ones, like edges or corners.

In the last years, deep neural networks have been successfully used in a variety of computer vision applications. They achieved outstanding results in image classification [28], object detection [12] or action recognition in videos [24], for instance. These networks have in common that they are discriminative models and map a high-dimensional input like an image to a class label. The aim of this models is to transform the input data into a representation where clear decision boundaries between the classes can be found. In contrast, deep generative models [13] learn the underlying structure of the input and how the data was generated. They try to model the actual data distribution, and can be used to generate new synthetic samples similar to existing data. The difference between these models is visualized in Figure 1.2. Formally, a discriminative model learns the posterior probability $P(y \mid \mathbf{x})$ over labels y given the data $\mathbf{x}$, and the generative model learns the probability distribution of the data for each class $P(\mathbf{x} \mid y)$ (likelihood function). The

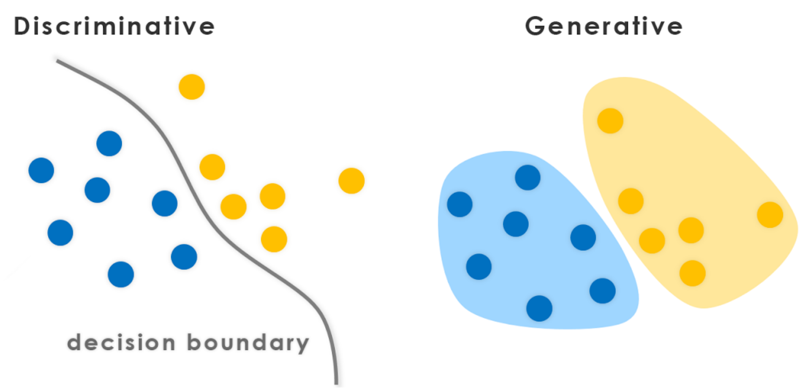


Figure 1.2: A discriminative model tries to find as clear as possible decision boundaries between the given classes. Aim of a generative model, on the other hand, is to model the actual data distribution of each class. Image taken from web source[1].

relation between these probability distributions is given by Bayes' theorem:

$$P(y \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y)P(y)}{P(\mathbf{x})}. \tag{1.1}$$

Compared to discriminative models, deep generative models did not reach this level of success until recently. Challenging are intractable computations, which arise during the maximum likelihood estimation or similar methods. Recent work introduced Generative Adversarial Networks (GANs) [14], a new approach to train generative models with neural networks. The idea is to pit two networks against each other in an adversarial game. The generator network tries to produce synthetic data that looks as real as that the discriminator network is not able to differentiate it from training data. This min-max game is completed, when real and synthetic data are indistinguishable for the discriminator. This approach sidesteps previous difficulties and performs very well on various datasets. However, the internal generation process from the initial noise vector to the resulting image is relatively unexplored. The aim of this thesis is to investigate the internals of *GANs* more deeply in order to better understand what they learn and how the generation process can be influenced to change semantic properties of synthesized samples. Moreover, the applicability of this model to solve two fundamental generative computer vision problems is analyzed.

1.2 Generative Models for Computer Vision

In this section, we will outline some applications of generative models in computer vision. Their abilities to capture the underlying generation process and to synthesize new samples are useful for many tasks.

[1]http://www.evolvingai.org/fooling Accessed: 21-01-2016

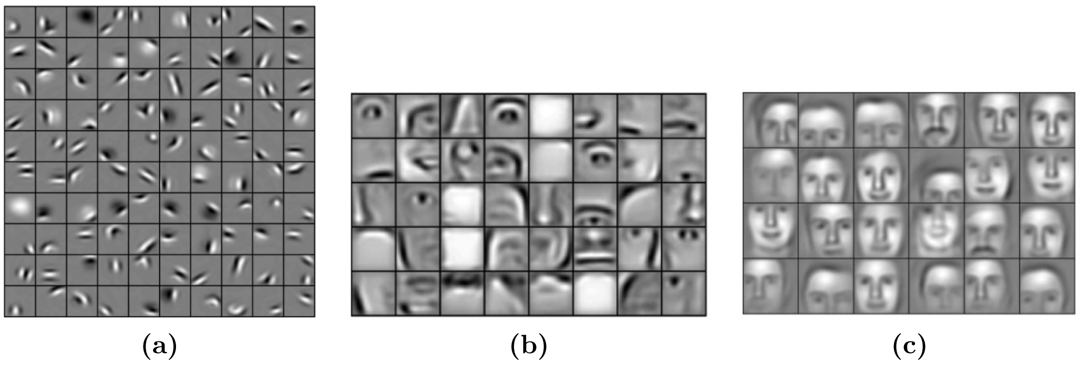


Figure 1.3: Learned hierarchical data representation for faces: Each of these features can be seen as a filter for particular structures within the input image. The first layer basically detects edges in the images (a), while the second layer (b) builds object parts, like a nose, based on the lower layer. The third layer (c) combines the parts of layer two into more complex features e.g. faces. Images are taken from [33].

1.2.1 Unsupervised Feature Learning

As previously described, the performance of a machine learning algorithm heavily depends on the data representation. Hand-crafted features are often designed for a particular application, do not generalize well and rely on expert knowledge. Instead, generative models can be used to learn reusable features from the data itself and automate the feature extraction. The possibility to train them in an unsupervised way is an additional gain of these models [5, 33]. In contrast to supervised learning, where human labor is necessary to annotate samples, data acquisition for unsupervised algorithms is much easier. They can benefit from the practically unlimited amount of unlabeled data and save the cumbersome manual annotation.

Deep generative models attempting to learn hierarchical data representations are of particular interest. Lower layers learn simple features and are then composed to model high-level structures. In this way, it is possible to learn complex data representations, that can then be used on a variety of supervised tasks, like object detection using a small amount of labeled data. An illustration of a hierarchical data representation for faces is shown in Figure 1.3.

1.2.2 Texture Synthesis

Textures are images with special characteristics. They describe a wide range of surfaces like plants, paper, minerals or wood. The images usually contain similar pattern repeated over and over again with a certain amount of random changes. The task of texture synthesis is, given a small image patch (texture sample), to create a new larger synthetic image with similar structural content. Similar in this context means, that for a human observer the new generated image describes the same surface. The typical application is to simulate

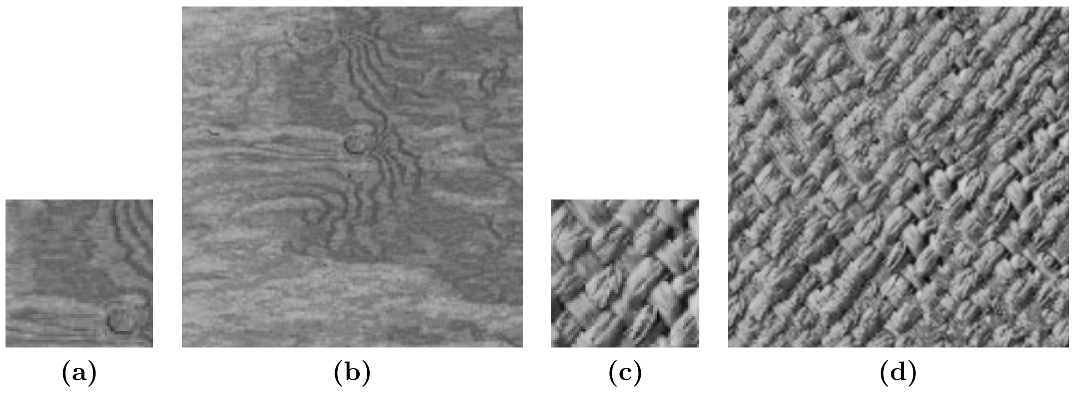


Figure 1.4: Texture synthesis on two real-world textures: (a) and (c) are the original texture samples, (b) and (d) are the larger synthesized images. Images are taken from [8].

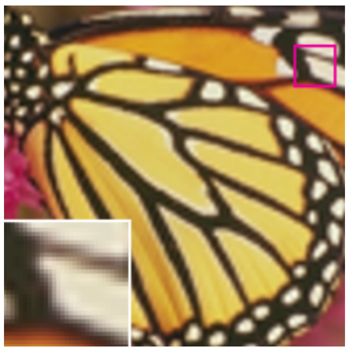
3D surfaces in video games or computer graphics by mapping a texture to objects, instead of explicitly modelling the surfaces.

Efros & Leung [8] for instance, perform texture synthesis by non-parametric sampling. They start with an randomly chosen 3x3 pixel seed from existing texture. For every new pixel, which borders already filled pixels, the algorithm finds similar patches in the source image by observing the neighboring pixels. Then one of these patches is randomly selected, and the center pixel is taken to be the newly synthesized pixel. Thereby, the process grows a new image one pixel at a time. Two examples of synthesized textures are shown in Figure 1.4.

1.2.3 Super-Resolution

To upscale an image, or increase the resolution of a given low-resolution image, is a fundamental operation in computer vision with high practical relevance. Many applications, like medical imaging or surveillance, need high-resolution images. However, this problem is inherently ill-posed, since for every low-resolution pixel, a vast number of possible solutions exist. An often used method to upscale a given image is bicubic interpolation. This method is very fast but leads to smooth edges and thus blurry high-resolution images. Here, a generative model can be used to learn the probability distribution of the latent high-resolution image, given the observed low-resolution image.

Recent super resolution approaches are usually example-based methods [48]. They learn a mapping from low- to high-resolution images, and use it to create the most-likely high-resolution version of a given input image. Dong *et al.* [6] for instance, train a deep convolutional neural network to learn an end-to-end mapping between low- and high-resolution images. An example of an upscaled image is shown in Figure 1.5.


(a)

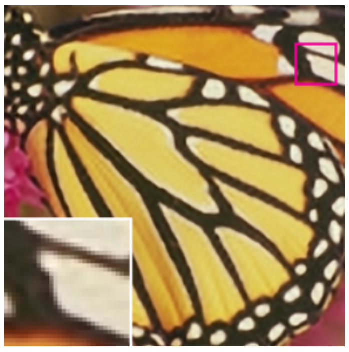

(b)

Figure 1.5: Super-resolution example: (a) is the low-resolution input image and (b) the synthesized high-resolution output. Images are taken from [6].

1.2.4 Inpainting

Old photographs often have damaged regions, like an aged corner or gaps through folds that should be recovered. Another time, we want to remove objects from our photographs, like a tower crane midst of a landscape. Inpainting, or scene completion, addresses these problems. The task is to fill or replace an image region, such that the modification is not visible for the human observer. A generative model can be used to synthesize new data likely to the surrounding image region.

Millions of photographs are used by Hays & Efros [16] to inpaint holes in images. They use a two-stage method to overcome computational and semantic challenges. Firstly they use a scene descriptor to find images in the dataset showing semantically similar content, and to reduce the amount of data from millions of images to a few hundred. Within these samples, they use pixel-wise Sum of Squared Differences (SSD) around the missing part, to find the best matching content for image completion. Two examples for inpainting are shown in Figure 1.6.

1.2.5 Dataset Augmentation

The amount of training data is a critical point for many machine learning applications, like image classification. Having not enough data can lead to a bad performance due to over-fitting, and increases the influence of noise or outliers. To acquire a larger training set is usually expensive and takes considerable effort. A generative image model can solve this problem by synthesizing new training samples and augmenting the existing dataset at little time and expense.

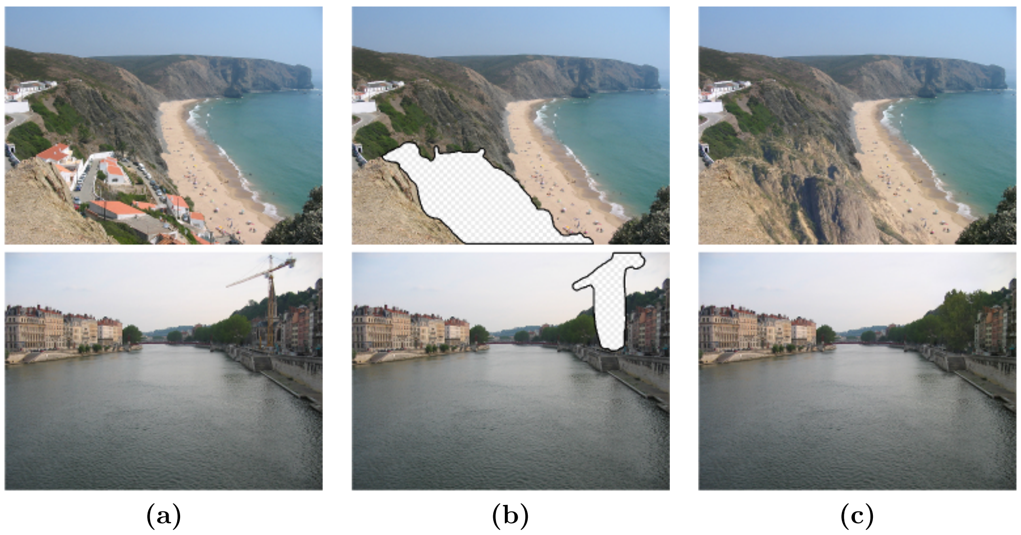


Figure 1.6: Example for inpainting: (a) is the original input image, (b) shows the image region that should be replaced and (c) is the synthesized output. Images are taken from [16].

1.3 Related Work

There are basically two approaches for generative image models: parametric and non-parametric models. The number of parameters in a non-parametric model grow with the amount of used training data. Examples for non-parametric models are shown in the previous section [8, 16]. Parametric models, like Generative Adversarial Networks (*GANs*), on the other hand have a finite number of parameters regardless of the observed data points.

The dominant approaches for parametric generative models are based on maximum likelihood estimation. The aim is to find a set of parameters $\boldsymbol{\Theta}$ of a probability model under which the data $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, $\mathbf{x}_i \in \mathbb{R}^d$ are most likely. The likelihood function at a single data point $\mathbf{x}_i$ is given by the probability density function at $\mathbf{x}_i$

$$P(\mathbf{x}_i \mid \boldsymbol{\Theta}). \tag{1.2}$$

Under the assumption that $\mathbf{x}_1, \dots, \mathbf{x}_N$ are independent and identically distributed (i.i.d.), the likelihood function for a set of points is given by the product over the individual likelihoods

$$P(\mathbf{X} \mid \boldsymbol{\Theta}) = \prod_{i=1}^{N} P(\mathbf{x}_i \mid \boldsymbol{\Theta}). \tag{1.3}$$

By taking the logarithm, products are replaced by sums which can be differentiated more easily. This does not change the optimization problem because the logarithm is a monotonic transformation. The log-likelihood is given by

$$L(\mathbf{X} \mid \mathbf{\Theta}) = \log(P(\mathbf{X} \mid \mathbf{\Theta})) = \sum_{i=1}^{N} \log P(\mathbf{x}_i \mid \mathbf{\Theta})). \tag{1.4}$$

Hence, the goal of maximum likelihood estimation is to find the set of parameters $\mathbf{\Theta}$ that maximizes the log-likelihood

$$\hat{\mathbf{\Theta}} = \arg\max_{\mathbf{\Theta}} L(\mathbf{X} \mid \mathbf{\Theta}). \tag{1.5}$$

1.3.1 Restricted Boltzman Machine

Generative models based on maximum likelihood estimation are Restricted Boltzmann Machines (RBMs) [10, 21, 50]. They are undirected graphical models, consisting of two layers and can be interpreted as stochastic artificial neural networks. They consist of a layer of visible and a layer of hidden neurons, and a fully symmetric connection between them. The difference to a Boltzmann machine is the restriction that the nodes form a bipartite graph, which means there are no connections within one layer. The structure of *RBMs* is shown in Figure 1.7. In terms of an image the visible nodes are image pixels and the hidden variables can be seen as feature representations. The *RBM* is an energy-based model [30], meaning it assigns a scalar energy to each configuration of variables. Learning means to modify the energy function such that plausible configurations have low energy.

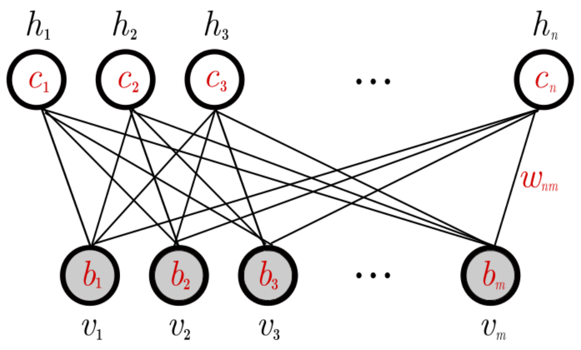


Figure 1.7: Structure of a restricted Boltzmann machine with m visible and n hidden nodes. Image taken from [10].

The joint probability distribution for a set of visible random variables $\mathbf{v}$ and hidden random variables $\mathbf{h}$ is given as

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})}, \tag{1.6}$$

with the energy function

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\intercal \mathbf{v} - \mathbf{c}^\intercal \mathbf{h} - \mathbf{v}^\intercal \mathbf{W} \mathbf{h}, \tag{1.7}$$

where **b** are hidden nodes biases, **c** visible nodes biases and **W** are the weights between the two layers. The partition function Z is a normalizing constant to ensure the distribution sums to one:

$$Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}. \tag{1.8}$$

The marginal distribution of the visible variables **v** is therefore given as the sum over all possible hidden node configurations:

$$P(\mathbf{v}) = \sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}. \tag{1.9}$$

Given a dataset of examples $\mathbf{V} = \{\mathbf{v}_1, \dots, \mathbf{v}_K\}$, the parameters of this model $\mathbf{\Theta} = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$ are estimated by maximizing the log-likelihood given as:

$$L(\mathbf{V} \mid \mathbf{\Theta}) = \log(P(\mathbf{V} \mid \mathbf{\Theta})) = \sum_{i=1}^{K} \log P(\mathbf{v}_i)). \tag{1.10}$$

Unfortunately, computing the exact gradient of this objective function is intractable because of the partition function. Since the number of possible configurations rise exponentially with the number of variables, it is computationally impossible to calculate the exact solution. These intractable partition functions are the major problem of this type of models. However, the common way to train *RBMs* is to approximate the log-likelihood gradient, for example using contrastive divergence [19].

1.3.2 Deep Boltzmann Machines

One possibility to build deep generative models are Deep Boltzmann Machines (DBMs) introduced by Salakhutdinov & Hinton [46]. They can be seen as a multi-layer extension of the Restricted Boltzmann Machine (*RBM*). Each layer captures higher-order correlations between the hidden features of the lower layer, and thereby the internal data representations become more complex.

In contrast to Deep Belief Networks (DBNs) [20], where *RBMs* are stacked and separately trained from bottom to top, a *DBM* is an entirely undirected graphical model where information from higher to lower layers is used during training. Even though, a greedy layer-by-layer pre-training is necessary to initialize the parameters. Figure 1.8 shows synthesized samples from a *DBM* trained on a 3D object dataset [46].

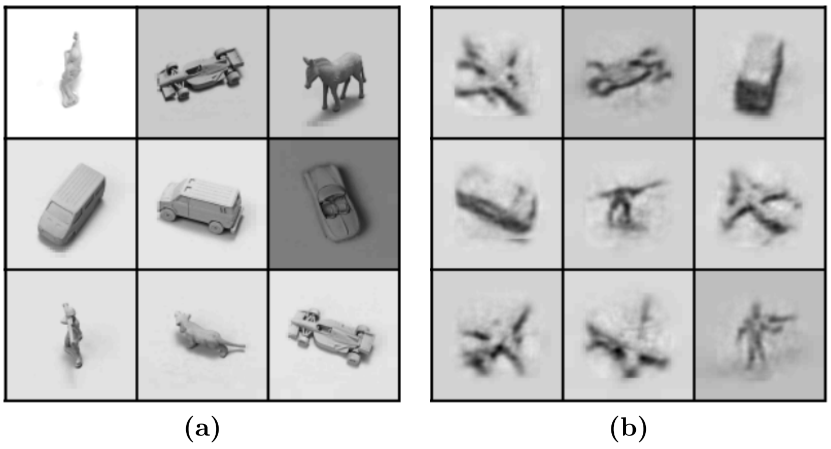


Figure 1.8: Synthesized samples from a Deep Boltzmann Machine (*DBM*): (a) shows random samples from the training set, and (b) are synthesized samples. Images are taken from [46].

1.3.3 Autoencoder

An autoencoder [21] is an artificial neural network used to learn a code representing the input. The simplest form consists of three layers: an input layer, a hidden layer representing the code and an output layer. The characteristic of autoencoders is that they are trained to reconstruct their own input, thus the number of nodes in the output layer is equal to the size of the input layer. The network can be seen as an encoder-decoder system. A typical application is dimensionality reduction, in which the number of nodes in the hidden layer is smaller than the number of nodes in the input and output layer. Therefore, the small layer in the center works like an information bottleneck. This typical structure of an autoencoder is shown in Figure 1.9. Thus, the model is forced to find the important properties within the data that shall be encoded. Another way to prevent the autoencoder from learning an identity function, is to add a sparsity penalty on the hidden layer (sparse autoencoder [32]).

Of particular importance for generative models are denoising autoencoders [59]. The idea behind this approach is to corrupt the input data, and let the network learn to reconstruct the uncorrupted version. Regarding images, the corruption could be to add white noise or to delete pixels. To revoke this process, the network has to learn the underlying distribution to capture the dependencies between the input neurons. These autoencoders can be stacked and trained layer by layer, to learn a deep data representation [60]. Further work explored methods to generate samples from a trained denoising autoencoder [1, 2]. Figure 1.10 shows samples generated by a denosing autoencoder trained on a handwritten digit dataset [2].

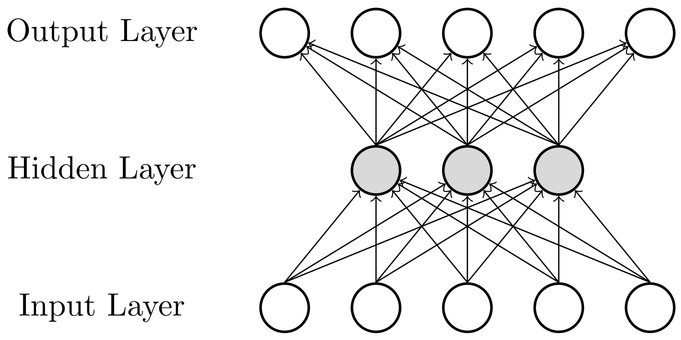


Figure 1.9: Typical structure of an autoencoder: The number of nodes in the input layer is equal to the size of the output layer. A smaller layer in the center works like an information bottleneck.

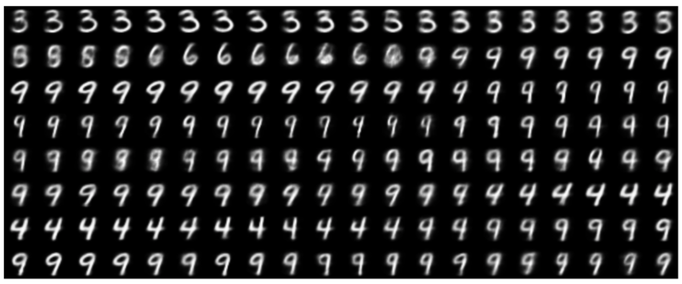


Figure 1.10: Samples generated by a denoising autoencoder, trained on a handwritten digit dataset. Image taken from [2].

1.4 Contribution & Outline

The aim of the first part of this thesis is to investigate the internals of *GANs*. The goal is to better understand what they learn and to find ways to manipulate the generation process in order to synthesize samples with desired properties. After an introduction to artificial neural networks and convolutional neural networks in Chapter 2, we describe the principles of *GANs* in Chapter 3, including recent extensions. In Chapter 4 we train adversarial nets on two common depth-datasets for head- & hand pose estimation and analyze the generator networks in a variety of ways. Furthermore, two ways to manipulate the generation process are presented, allowing the creation of depth-data for desired head- or hand-poses.

In the second part of this work, the applicability of *GANs* to solve two fundamental computer vision problems is analyzed. In the first experiment in Chapter 5 we show that learned discriminator networks are a strong candidate for unsupervised feature learning. The second experiment demonstrates that *GANs* can be used for super resolution to significantly increase the quality of upsampled images.

2

Artificial Neural Networks

Contents

The aim of this Chapter is to introduce the principles of artificial neural networks on which Generative Adversarial Networks (GANs) are based. In the first part we will outline the development of neural networks, discuss the basic concepts like neurons or layers and describe the training process. In the second part, we focus on convolutional neural networks and show their advantages for computer vision tasks.

2.1 Feedforward Neural Networks

As their name suggests, artificial neural networks [13, 43] are inspired by biological neural systems. The mammalian brain ability to adapt to different tasks using the same structures motivates this field of artificial intelligence research. Von Melchner *et al.* [61] for instance, found that ferrets can learn to see with their auditory area of the brain, if retinal projections were rewired. This suggests, that the brain is able to solve different tasks using the same learning approach. Artificial neural networks originally attempt to model this learning process based on neurons. Nowadays, the research in artificial neural networks diverged from neuroscience, since we do not have enough information about the activities of neurons in the mammalian brain to model them. Another reason is, that it is not guaranteed that the biological model always leads to the optimal solution. Therefore, it becomes more a topic of engineering, and a general approach to learn multiple levels of representation. However, the basic idea to solve complex tasks using the interaction between many simple computational units is inspired by neuroscience. Therefore, we begin this section with a short explanation of a biological neural system.

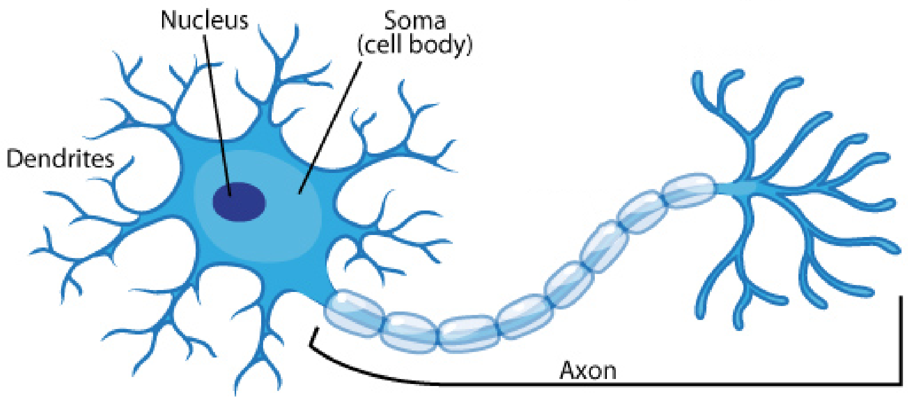


Figure 2.1: Illustration of a biological neuron: Each neuron has several inputs (dendrites) and a single output (axon) connected to dendrites of other neurons. Image taken from web source[1].

2.1.1 Biological Neural Network

The core component of the brain is called neuron or nerve cell [62], illustrated in Figure 2.1. This unit receives signals from other neurons and produces an output signal. Each neuron consists of several input channels called *dendrites*, and a single output called *axon*, which may split up into several branches. Neurons are connected to dendrites of other neurons by *synapses*. Each of the synapses possesses a synaptic weight, indicating the strength of the connection. Inputs, which can be excitatory or inhibitory, depending on their weight, are carried toward the cell body where they are summed up. If the sum is greater than a threshold, the neuron generates a brief pulse along its axon to stimulate other neurons, also referred to as *firing*. The interconnection of a large amount of these neurons forms the biological neural network. The important knowledge within such a network, is represented by the synaptic weights between the neurons. This is a very coarse model of a biological neuron, with several simplifications. For instance, the synapses in a real neuron are not just a single weight, but a complex chemical system. However, this model shows the basic biological concepts that inspired the development of artificial neural networks.

2.1.2 Artificial Neurons

The first artificial neuron was proposed by McCulloch and Pitts in 1943 [37]. In this work, the neural activity is modeled as a threshold unit over a weighted sum of inputs. The parameters of this model needed to be set manually. In 1958, Frank Rosenblatt [44] extended this work, and introduced with the Perceptron a model that was able to learn the parameters using an iterative learning algorithm. This work represents the base for modern artificial neural networks.

Similar to a biological neuron, the computational model receives inputs from other

[1]http://biofoundations.org/?p=3283 Accessed: 31-01-2016

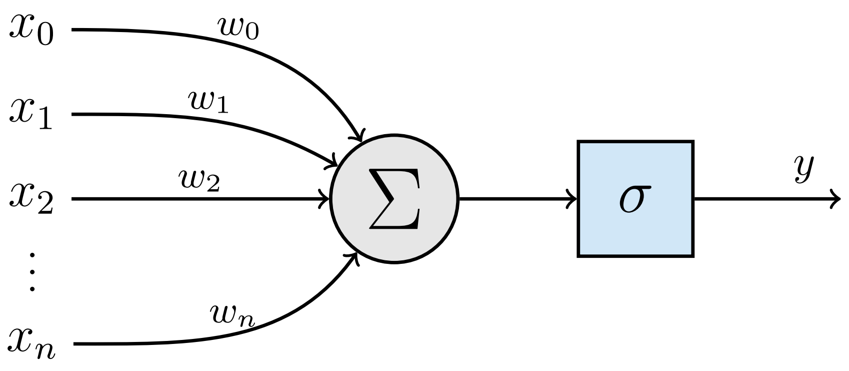


Figure 2.2: Illustration of an artificial neuron: The inputs x_i are weighted with w_i, summed up, and passed through the activation function to create the output y.

neurons, combines them in some way and outputs the final result. It consists of several inputs x_i, representing the dendrites. Each of these inputs is weighted by w_i, equal to the synaptic weights. These learnable parameters control the intensity and direction of influence (excitatory or inhibitory) between the neurons. Usually, the input x_0 is set to 1 modeling a bias input. This weighted sum over the inputs is then passed through a transfer or activation function $\sigma(\cdot)$, to create the output y which corresponds to the axon in a biological neuron. The structure of an artificial neuron is shown in Figure 2.2. This means, each neuron calculates the dot product between the inputs $\mathbf{x}$ and the weights $\mathbf{w}$, and applies the activation function:

$$y = \sigma\left(\sum_i w_i x_i\right) = \sigma\left(\mathbf{w}^\intercal \mathbf{x}\right). \tag{2.1}$$

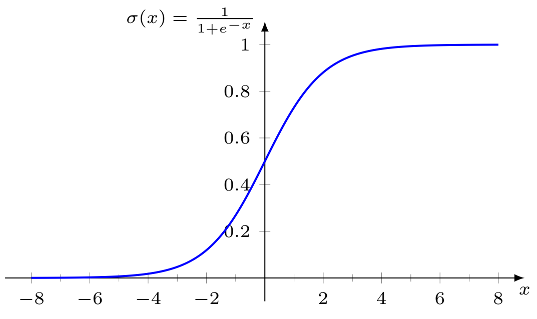
Learning the parameters $\mathbf{w}$, given a set of input data and corresponding target values t, means to solve an optimization problem. The loss function E, determines the error between a target t and the output y, for instance:

$$E = \frac{1}{2}(t-y)^2 = \frac{1}{2}(t-\sigma\left(\mathbf{w}^\intercal \mathbf{x}\right))^2. \tag{2.2}$$

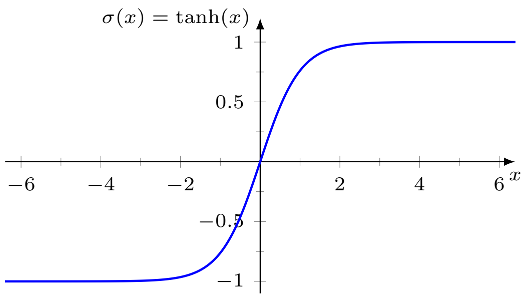
The goal is to find the set of parameters $\mathbf{w}$, that minimize the loss function. This is usually done by gradient descent, where the parameters $\mathbf{w}$ are randomly initialized, and iteratively refined until the loss is minimized. The direction of the update is given by the gradient of the loss function. The update rule for the parameters is therefore

$$w_i^{t+1} = w_i^t + \Delta w_i = w_i^t - \alpha \frac{\partial E}{\partial w_i}, \tag{2.3}$$

where α is the step size or learning rate.



(a)



(b)

Figure 2.3: The sigmoid activation function (a) maps real numbers into a range from 0 to 1. Similarly, the hyperbolic tangent non-linearity (b) maps x to [-1,1].

2.1.3 Activation Functions

The first artificial neurons, like ADALINE [63], used linear activation functions like the identity function. Their problem was, that a network consisting of these linear neurons, remains a linear function over its inputs and has no benefit. This means, that a multi-layer neural network of linear neurons, can always be replaced by a single layer network. Therefore, it is the non-linearity, or the capability to learn non-linear representations, respectively, that makes artificial neural networks so powerful. Another important property for an activation function besides the non-linearity is, that it has to be differentiable or at least sub-differentiable. This is necessary to use sub-gradient based optimization methods.

2.1.3.1 Sigmoid

A historically often used activation function is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$. It maps a real-valued number into a range from zero to one, see Figure 2.3a. The output can be seen as the probability, that a neuron is firing, equal to the firing rate in biological neurons. A major disadvantage of this function is, that it saturates across most of their domain. Small values of x converge to zero and large values to one. Therefore, the gradients in this border areas are almost zero and make gradient based optimization very difficult.

2.1.3.2 Hyperbolic tangent

A similar activation function is the hyperbolic tangent, shown in Figure 2.3b. It maps a real valued input to the range from minus one to one. Although this activation function saturates similar to the sigmoid, it performs better because the output is zero centered ($\tanh(0) = 0$) [31].

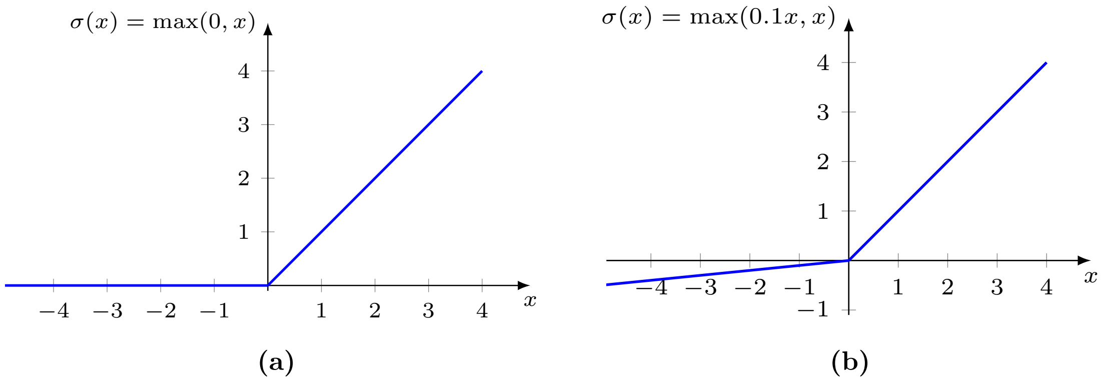


Figure 2.4: The Rectified Linear Unit (*ReLU*) is zero for vallues smaller than zero and equal to the identity function for values greater than zero (a). To guarantee a valid gradient also for negative values, the Leaky Rectified Linear Unit (*leaky ReLU*) has a small negative slope (b) for $x < 0$.

2.1.3.3 ReLU

An often used activation function in the last years is the Rectified Linear Unit (ReLU) [39] $\sigma(x) = \max(0, x)$, shown in Figure 2.4a. It is similar to the identity function used in a linear neuron except that the output for negative values is zero. Due to this non saturating linear form, the derivatives are large whenever the unit is active. This solves the gradient vanishing problem in sigmoid or hyperbolic tangent functions. These large gradients lead to a faster learning process. For instance, Krizhevsky *et al.* [28] showed that a neural network for image classification trained six times faster using *ReLUs* compared to hyperbolic tangent activation functions.

A drawback of this activation function is the *dying ReLU problem* [15]. This occurs, when a large enough gradient changes the parameters such that the neurons output is equal to zero for all training samples. From that point, the gradients will always be zero and the unit can not be reactivated.

2.1.3.4 Leaky ReLU

The Leaky Rectified Linear Unit (leaky ReLU) $\sigma(x) = \max(\alpha x, x)$ [36], shown in Figure 2.4b, addresses the previously described *dying ReLU problem*. For values smaller than zero, a *leaky ReLU* has a small negative slope adjusted with α. This guarantees a small gradient for negative inputs.

There exist several variations of this activation function, like the PReLU [18] where the slope value is a learnable parameter. Other variants are the RReLU [64], where this value is randomly chosen, or the ELU [3] which employs an exponential saturation function as negative part.

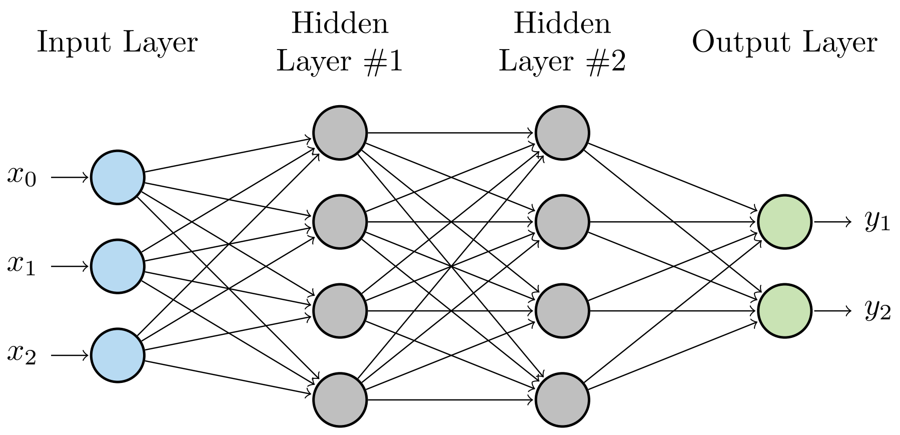


Figure 2.5: Illustration of a multilayer feedforward neural network: To evaluate this network, the data is processed in one direction, from the input layer over the hidden nodes to the output layer.

2.1.4 Multilayer Neural Networks

A single artificial neuron as previously described, is a very simple model, computing a non-linear input-output mapping. But, by composing them in a multilayer architecture, they are able to learn very complex functions. This networks are called *feedforward neural networks* or *multilayer perceptron.* A simple example for the architecture of this networks is shown in Figure 2.5. The neurons are grouped into layers and connected in an acyclic graph, where the output of a neuron becomes an input to other neurons, without building cycles. Concretely, each neuron of one layer is connected to all neurons of the subsequent layer. Neurons within a single layer do not have any connections. This type of layers are also known as *fully connected layer.*

The first layer, the leftmost layer, is called *input layer* where the number of neurons is equal to the dimension of the input data. On the opposite side of the graph, the rightmost layer, is the *output layer.* In a classification task for instance, the number of neurons in this layer is equal to the number of classes. The layers in between are called *hidden layers*, because they do not have any connections to the outside. These networks are called feedforward, because the information moves only in one (forward) direction. Starting at the input layer, going through the hidden nodes to the output layer, without any loops.

One advantage of this network architecture is, that the forward path is very efficient to compute using simple matrix vector multiplications. Consider the network shown in Figure 2.5. The input can be represented as a vector $\mathbf{x} \in \mathbb{R}^{3\times 1}$. All weights of the first hidden layer can be stored in a matrix $\mathbf{W^1} \in \mathbb{R}^{4\times 3}$, with row i representing the weights of the i-th neuron. The output $\mathbf{h^1} \in \mathbb{R}^{4\times 1}$ of the first hidden layer is then given as the result of the matrix vector multiplication

$$\mathbf{h^1} = \sigma(\mathbf{W^1}\mathbf{x}), \tag{2.4}$$

where $\sigma(\cdot)$ is an arbitrary activation function. In the same way, $\mathbf{h^2}$ and the network output $\mathbf{y}$, can be computed. An important property is, that instead of a single input data point, a whole batch of data can be passed through the network at once. In this case, the input is a matrix $\mathbf{X} \in \mathbb{R}^{3\times d}$, where each column represents a data point.

2.1.5 Gradient Based Learning

So far we described the architecture of multilayer neural networks and how the output for a given input-sample is computed. Now, we are going to show how such a network can be trained. The idea is to define a loss function measuring the network performance, and minimizing this cost by iteratively adapting the network weights. The direction of the parameter update is given by the gradient of the loss function.

Forward Propagation

The first step is the already described *forward propagation*, where for a given sample $\mathbf{x}$ the information is propagated from the input layer over the hidden layers to the output nodes. Now, an error- or objective function $E(\mathbf{W})$ is used to measure the difference between the network output $\mathbf{y}$ and the desired output $\mathbf{t}$, resulting in a scalar cost, for instance using the sum of squared differences (Equation 2.2). Minimizing this loss by adapting the network weights is the goal for the subsequent steps.

Backward Propagation

In order to minimize the objective function, it is necessary to compute the gradient of the cost function with respect to the parameters $\partial E/\partial w_i$. It would be possible to compute an analytical expression for the gradient, but numerically evaluating this expression is computationally expensive and inefficient. For instance, several sub-expressions may be used repeatedly and either needed to be stored or recomputed several times. This problems are addressed by the *backpropagation* algorithm [45], which is based on the chain rule for derivatives. The main principle is, that the gradient of the error function with respect to the input weight of a neuron can be calculated using the gradient with respect to the output of that neuron. This gradient is given by a weighted sum over the input gradients of the neurons in the layer above. In this manner, the error can be back-propagated through the network from the output layer to the input layer. This error backpropagation is shown in Figure 2.6.

More formally this means if o_j is the output of neuron j, and a_j is the sum over all inputs for this neuron, then $o_j = \sigma(a_j)$, where $\sigma(\cdot)$ is an arbitrary activation function. The gradient of the objective function with respect to weight i for neuron j is then based on the chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}}. \tag{2.5}$$

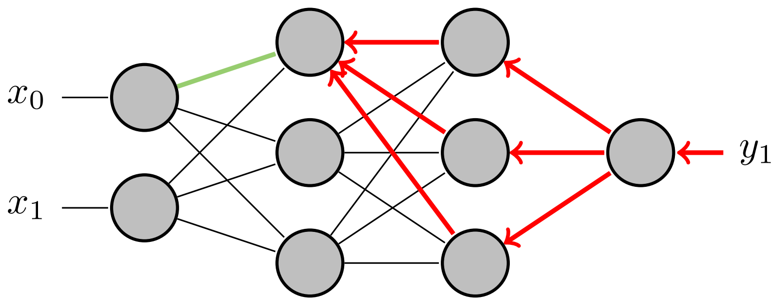


Figure 2.6: Error Backpropagation: The error of the objective function is back-propagated (red) from the output layer to compute the gradients of weights in previous layers (green).

The last term is the derivation of the sum over inputs with respect to weight w_{ij}

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{k=1}^{n} w_{kj} o_k \right) = o_i, \tag{2.6}$$

where o_i is the relevant output of the previous neuron. The second term is the derivation of the activation function and the first term represents the impact of the current neurons output to the error function. If the neuron is an output neuron, this is simply the derivation of the loss function, otherwise it is given as

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial a_l} \frac{\partial a_l}{\partial o_j} \right), \tag{2.7}$$

where L are the neurons in the next layer using o_j as input. Because of the error back-propagation, the gradients with respect to o_l are already known.

Gradient Descent

The gradients computed in the previous backpropagation step indicate the direction towards an error function maximum. In order to minimize the loss, *gradient descent* updates the parameters in the negative direction of the gradient

$$w_{t+1} = w_t - \alpha \nabla_w E(w_t), \tag{2.8}$$

where α is the learning rate or step-size, and $\nabla_w E(w)$ are the gradients of the error function with respect to the parameters. This process is visualized in Figure 2.7.

The usually used gradient descent implementation is *mini batch gradient descent*, where B samples out of the training set are used to compute the gradients. B is called *batchsize* and is an important parameter. The advantages are less computational effort updating the weights and efficient matrix $\times$ matrix multiplications to compute several samples

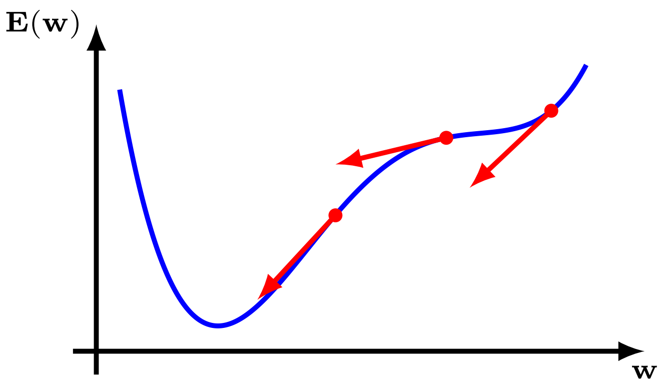


Figure 2.7: Gradient descent: In order to minimize the error function E, gradient descent take steps in direction of the negative gradient, until the minimum is reached.

simultaneously. This method is sometimes also called *stochastic gradient descent*, even though this variant usually uses only a single example ($B = 1$) in one step.

However, there is no guarantee that these methods converges to the optimal solution. Choosing a proper learning rate for instance, heavily influences the process. A too small step-size leads to very slow convergence, while a too large learning rate can cause a fluctuation around the minimum or to divergence. Another challenge is, that the error function is usually non-convex, which means there can be many local minima besides the global minimum where the algorithm can converge to. Furthermore, saddle points can lead to gradients close to zero and stop the optimization. In the following, we will outline extensions to the stochastic gradient descent algorithm addressing these problems.

The first one is the *momentum* method [45], which prevents from oscillating gradients. The update rule is given as

$$v_{t+1} = \mu v_t - \alpha \nabla_w E(w_t) \tag{2.9}$$

$$w_{t+1} = w_t + v_{t+1}, \tag{2.10}$$

where μ is the momentum parameter. If the error surface has the form of a narrow valley with steep sides, the gradients tend to oscillate across these sides. To suppress this behavior, the method incorporates the previous gradients into the current update.

Nesterov's Accelerated Gradient [40] is a similar method, but the gradients are calculated with respect to the approximated future position:

$$v_{t+1} = \mu v_t - \alpha \nabla_w E(w_t + \mu v_t) \tag{2.11}$$

$$w_{t+1} = w_t + v_{t+1}. \tag{2.12}$$

The approximation is done using the previous gradients. Therefore, the algorithm can detect an upcoming increase and react earlier.

Adagrad [7] adapts the learning rate for each parameter based on the gradients history for that parameter. The rescaling is done by dividing each gradient by the square root of the sum over historical gradients. It reduces the step-size for parameters with constantly large gradients and increases the learning rate for infrequent parameters. The advantage is a larger weighting of rare but maybe useful features. A similar method is *RMSProp* [54], where the learning rates are rescaled using a moving average of the squared gradients for each weight. The recently proposed *Adam* algorithm [26] can be seen as a combination of RMSProp and the momentum method. It also computes adaptive learningrates using averaged squared gradients like RMSProp, but in addition, it also keeps an average over past first order moments for smoothing the gradients similar to momentum.

2.1.6 Regularization

One problem using artificial neural networks is overfitting. It occurs especially when the number of training samples is small compared to the network complexity. It is characterized by a very small training error but a large test error. This means, that the network has memorized the training data, but is not able to generalize to new data. There are several possibilities to prevent overfitting as described in the following.

2.1.6.1 Weight Decay

Weight decay, or L2 regularization is an often used technique to avoid overfitting. The idea is to add a regularization term to the objective function:

$$E_R = E + \frac{\lambda}{2} \sum_w w^2. \tag{2.13}$$

This term, the sum over all squared network weights, penalizes weight vectors with single peaks and prefers small scattered weights. Smaller weights mean lower network complexity and provide a simpler data representation. The strength of the regularization is determined by the parameter λ. A similar method is L1 regularization, where in the regularization term $|w|$ is used instead of w^2. This leads to sparse weight vectors, where many weights are close to zero. Therefore, the network has to concentrate on a few important connections.

2.1.6.2 Dropout

A recent method to prevent overfitting is dropout [52]. It does not modify the objective function like weight decay, instead it modifies the network architecture. The idea is to randomly drop neurons during the training process in hidden layers, as shown in Figure 2.8. Dropping means setting the outputs of these neurons to zero. Therefore, the neurons can not rely on the existence of other neurons, and are forced to learn a more robust data representation. A network, trained in this manner, can be seen as an ensemble of many sub-networks sharing the same parameters. The number of active neurons is determined

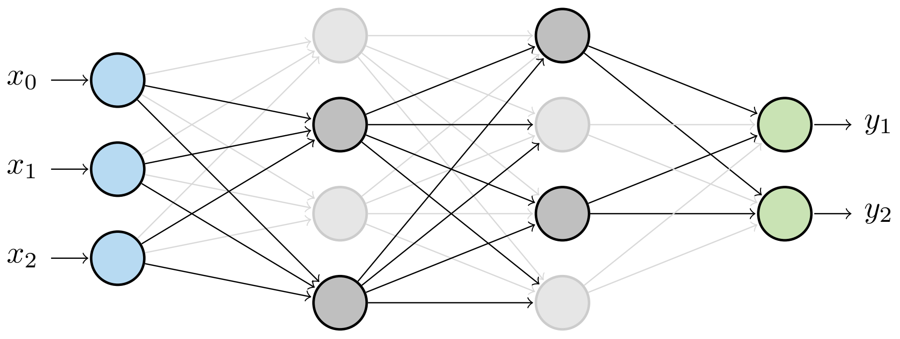


Figure 2.8: An example for a thinned neural network by applying dropout to the hidden layers.

by a probability parameter p for each hidden layer. In each forward - backward iteration, a new subset of neurons is chosen.

2.1.6.3 Batch Normalization

A problem in deep neural networks is, that the input data distribution of a layer changes as the weights of the previous layers change. The common way to suppress this effect is to use small learning rates and carefully initialized parameters. Another approach, proposed by Ioffe & Szegedy [23], addresses this problem by normalizing the layer inputs. If the inputs to each layer within one patch have zero mean and unit variance, the learning process can by stabilized even for higher learning rates.

2.2 Convolutional Neural Networks

The history of Convolutional Neural Networks (CNNs) begins with experiments of Hubel & Wiesel [22] in the 1960s. They were studying the visual cortex of cats. In particular, they investigated an early visual area by recording neural activity while the cat was looking at different patterns on a screen. The found that neighboring cells in the visual cortex, process neighboring regions on the retina, which means that the spatial information is preserved within the brain. In addition, they figured out that there exists a hierarchical organization. Simple cells, responding to particular orientation of edges in their region of the visual field, transmit their information to complex cells. These cells respond to the same orientation of edges, but in a larger region. All this suggests a hierarchical structure, where cells are build on top of each other, from locally receptive fields to complex representations. Modelling these concepts with artificial neural networks is the motivation for this type of architecture.

The difference between Convolutional Neural Networks (*CNNs*) and conventional neural networks is, that *CNNs* are designed to process data with a known grid-like organization like images [13]. This knowledge can be exploited to adapt the network architecture, such that the spatial information within the input data can be preserved and used through the computation. In a fully connected neural network, the input layer, as well as the output of a subsequent layer, is represented by a single vector. In contrast, *CNNs* operate over arrays, where the data are arranged in three dimensions, described by width, height and channel. A RGB color image as input volume for example, has three channels, consisting of 2D pixel arrays. All layers in a *CNN* take arrays as input, and output arrays of activations, compared to vectors in fully connected networks. Connecting all neurons in one layer to all neurons of the previous layer is impractical, when dealing with high-dimensional inputs like images. Therefore, the neurons within a *CNN* process only small input regions, similar to the receptive fields. But, if a neuron is able to detect a specific type of feature at one location in the input, then the same weights are also useful at a different location. This leads to parameter sharing, where neurons within one slice use the same weights. This important constraint dramatically reduce the number of parameters within the network. Nonetheless, *CNNs* consist of neurons with learnable parameters and a non-linear activation function, as well as conventional neural networks. They can be trained using the same principles by minimizing a scalar cost function. Therefore, all techniques described in the previous section can also be used for *CNNs*.

2.2.1 Convolutional Layer

2.2.1.1 Overview

The convolutional layer is the key element for this type of networks. As already mentioned, the input to a convolutional layer is a data array. The learnable parameters or weights of this layers are a group of filters, also represented as arrays. These filters width and height is usually small, compared to the input array, but they are always equal in the number of channels. During the forward propagation, each of these filters is convolved over the input array along its width and height. Convolving means to slide the filter spatially over all possible locations in the input, and compute the dot-product between input section and the filter repeatedly. This results in a two dimensional output, called feature-map, representing the filter responses at every spatial location. An example for the convolution using a single filter is shown in Figure 2.9a. Every learned filter is sensitive to a particular type of structure in the input. Therefore, a high value in the activation map is a sign for the presence of this feature at the corresponding input location. For multiple filters, the resulting feature-maps are stacked along the channel axis, forming the output array. Therefore, the neurons arrangement in a convolutional layer can be seen as a three-dimensional volume, compared to a single vector in fully connected layers. An example for a convolution layer with three filters is shown in Figure 2.9b.

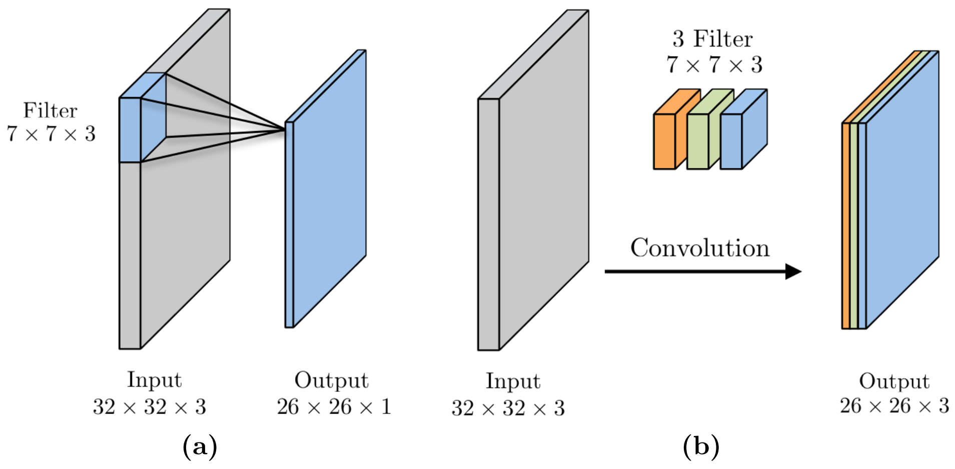


Figure 2.9: (a) shows an example for a convolutional layer with one filter. By sliding the filter over all possible locations in the input and computing dot-products between the input slice and the filter weights, an output feature-map is created. If multiple filters are used, the resulting output array is created by stacking the feature-maps along the channel dimension (b).

2.2.1.2 Convolution Operation

Formally, the convolution between a two-dimensional input $\mathbf{X}$ and a filter kernel $\mathbf{F} \in \mathbb{R}^{(2k+1)\times(2k+1)}$ is given as

$$\mathbf{Y}(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} \mathbf{X}(i-u, j-v)\mathbf{F}(u,v). \tag{2.14}$$

Though, most neural network frameworks implement instead a related operation called cross-correlation, given as

$$\mathbf{Y}(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} \mathbf{X}(i+u, j+v)\mathbf{F}(u,v), \tag{2.15}$$

but call it convolution. The difference to a convolution is that the kernel is not flipped during the cross-correlation computation. Thus, the cross-correlation is not commutative which usually does not matter for the implementation. An example for a two-dimensional convolution is shown in Figure 2.10. It shows that the convolution can be computed as dot-products between the input slices and the filter kernel. The 2D convolution can easily be extended to volumes by simultaneously performing 2D convolutions over all channels and summing up the results.

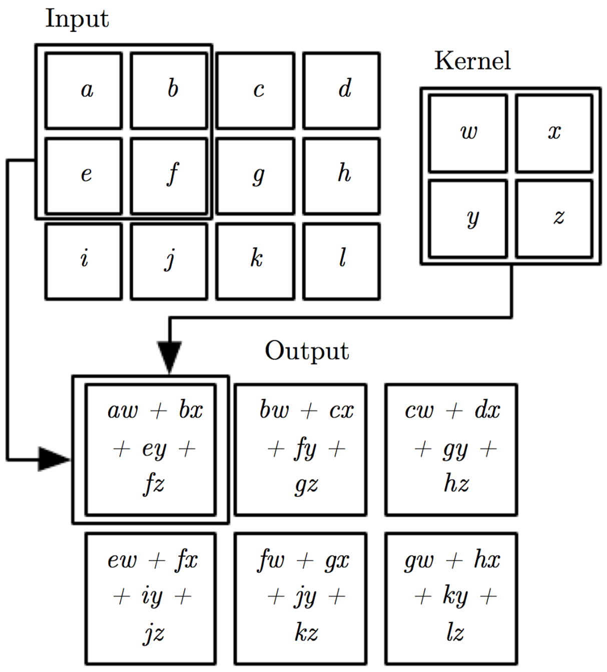


Figure 2.10: Example for a two-dimensional convolution. Each output element is computed as sum over the elementwise multiplication between the input slice and the filter kernel. Image taken from [13].

2.2.1.3 Shared Weights

As already mentioned, the neurons arrangement in a convolutional layer can be seen as a volume. Each of these neurons processes only a small input region by computing the dot-product with its filter coefficients, as shown in Figure 2.9a. The idea behind parameter sharing is the following: If a filter is able to detect a specific type of feature, like a horizontal edge, at one location in the input, then this filter can also be used at a different location. Therefore all neurons within one channel (one slice of the volume) share the same filter weights to reduce the number of parameters. This assumption is also necessary to compute the forward propagation as convolution between the input and the filters. Each channel in the resulting feature-map can be seen as response of the input to the given filter. To compute the gradients for the filter weights using the backpropagation algorithm, the gradients for each neuron are determined separately, and then summed up across each channel.

2.2.1.4 Parameters

Consider an input of size $W_{in} \times H_{in} \times C_{in}$, describing width, height and number of channels of the input data array. The dimensions of the output $W_{out} \times H_{out} \times C_{out}$, depend on four hyperparameters:

Number of Filters

The convolution between a single filter and the input results in one channel of the output feature map, see Figure 2.9b. Therefore, the number of output channels C_{out} is determined by the number of used filters N. This parameter is usually a power of 2, ranging from 32 to 512, and increases from early layer to the output. It can also be seen as the number of neurons that process the same input area using different weights.

Filter Size

The filter dimensions determine the spatial input region for each neuron and can be seen as the size of the receptive field. As already mentioned, the number of filter channels is always equal to the number of input channels C_{in}. This means the connections are local for width & height, but always cover all input channels, see Figure 2.9a. Since the filters are always quadratic along width and height, the filter size is given as a single parameter F. To ensure a definite filter center, the filter size is usually odd-numbered. The convolution is only defined where the input array and the filter are fully overlapping. This is not the case for the input edges, yielding to an output array with smaller width and height than the input. Therefore, the output dimensions are given as $W_{out} = W_{in} - F + 1$ and $H_{out} = H_{in} - F + 1$. Karen Simonyan and Andrew Zisserman [49] showed that it might be beneficial to use multiple layers with very small receptive fields, instead of on layer with a large filter size. First, they argue that incorporating three non-linearities instead of a single one, leads to a more discriminative decision function. The second advantage is that the number of parameters is decreased. For instance, three 3×3 convolution layers processing C input channels consist of $3(3^2C^2) = 27C^2$ weights. At the same time, a single convolution layer using 7×7 filters would require $7^2C^2 = 49C^2$ parameters, while the effective receptive field size remains the same.

Padding

A possibility to prevent the spatial reduction is to symmetrically pad zeros around the input. The padding parameter P, determines the number of rows or columns added on both borders, leading to $W_{out} = (W_{in} + 2P - F) + 1$ and $H_{out} = (H_{in} + 2P - F) + 1$.

Stride

The last parameter S, is called stride and defines the shift of the filter along width and height at each convolution step. It can be seen as the distance between the receptive field centers of two nearby neurons. Therefore, a stride of 1 means that the filter heavily overlap. In contrast, higher strides will lead to larger gaps between the receptive fields and to smaller outputs, where $W_{out} = (W_{in} + 2P - F)/S + 1$ and $H_{out} = (H_{in} + 2P - F)/S + 1$.

In summary, the convolution layer has four parameters: The number of filters (N),

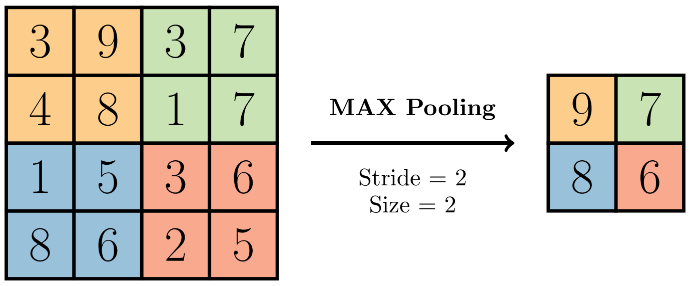


Figure 2.11: MAX pooling with region size equal to 2 and a stride of 2. Each sub-region is replaced by its maximum value to reduce the data dimensionality.

the filter size (F), a stride parameter (S) and the amount of input padding (P). The dimensions of the output array are then given as:

$$W_{out} = \frac{W_{in} + 2P - F}{S} + 1 \tag{2.16}$$

$$H_{out} = \frac{H_{in} + 2P - F}{S} + 1 \tag{2.17}$$

$$C_{out} = N \tag{2.18}$$

2.2.2 Pooling Layer

In a typical *CNN* architecture, a convolutional layer is followed by a non-linear activation function. The subsequent element is usually a non-linear downsampling layer, called *pooling layer*, before the next convolutional layer is applied. The purpose of this downsampling is to make the features invariant to small transformations and to reduce the data dimensionality. It replaces the data at a particular location using statistics of neighboring data. The data array is divided into small non-overlapping rectangles along width and height. For each rectangle, the output is only a single value. In MAX-pooling for instance, the output is the maximum value within the sub-region. Another variant is MEAN-pooling, where the the mean value is used to represent the rectangle. Important is that the pooling operation is applied on each channel separately, therefore the number of channels remains the same. The parameters of a pooling layer are a stride, representing the distance between the rectangles, and the size of the sub-region. Usually, the stride is even-numbered and equal to the rectangle size to ensure non-overlapping regions. An example for MAX pooling is shown in Figure 2.11.

2.2.3 Network Architecture

A typical Convolutional Neural Networks (*CNN*) architecture for classification consists of several blocks of convolutional and ReLU layers followed by pooling layers. These blocks are stacked on top of each other until the spatial dimension of the output array is small enough to use them as input to a fully connected layer. Usually, the number of feature maps is increased while the spatial dimension is decreased from stage to stage. The output of the last fully connected layer represents the class scores. Note that only convolutional layers and fully connected layers contain learnable parameters. The ReLU, as well as the pooling layers perform fixed operations. To control the network complexity, all regularization methods described for conventional neural networks can be applied to these networks as well. For instance, dropout is often used in the last fully connected layers to prevent overfitting.

A famous architecture is the *AlexNet* from Alex Krizhevsky *et al.* [28], shown in Figure 2.12. It won the Imagenet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 and draw attention to convolutional neural networks. It classifies natural images into one thousand different classes. The network contains approximately 650,000 neurons and 60 million learnable weights. It consists of five convolutional and three fully connected layers. Three pooling layers in-between are used to reduce the data dimensionality. It was the first network that used ReLU as non-linear activation function. For regularization, they used dropout in the last fully connected layers.

A more recent architecture is the *VGGNet*, proposed by Karen Simonyan and Andrew Zisserman [49] in 2014. They used only 3×3 convolution layers in combination with 2×2 pooling layers in their network, and demonstrated that increasing the depth is beneficial for the network performance. In the same year, Szegedy *et al.* [53] won the *ILSVRC* with their convolutional network called *GoogLeNet*. They introduced an *Inception Module*, that significantly reduces the number of weights in the network. Instead of performing a single convolution with a fixed kernel size, they us filters of several sizes to capture features at different scales and concatenate the results. Moreover they use average pooling layers instead of fully connected layers at the top of the network to reduce the amount of parameters. In 2015, the *ILSVRC* was won by *ResNet*, a architecture proposed by Kaiming He *et al.* [17]. They use a very deep residual network consisting of 152 layers. The idea is to include skip-connections that shortcut a few convolutional layers, providing a clear path for the gradients to reach early layers within the network during the error back-propagation.

[2]http://www.cc.gatech.edu/hays/compvision/proj6/deepNetVis.png Accessed: 17-02-2016

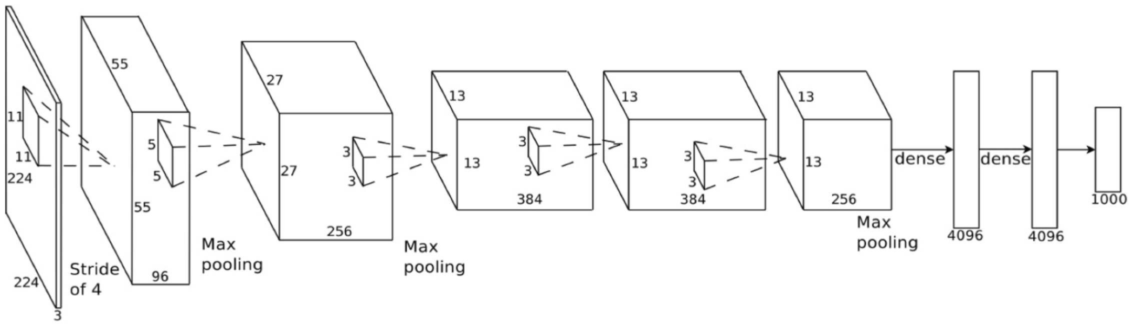


Figure 2.12: Alexnet: Convolutional neural network architecture to classify natural images into 1000 different classes. Image taken from web source[2].

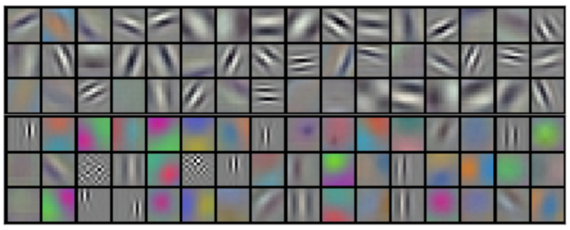


Figure 2.13: 96 learned filters ($11 \times 11 \times 3$) in the first convolution layer of Alexnet. Image taken from [28].

2.2.4 Visualization

Visualizing the internals of convolutional neural networks is useful to get an intuitive understanding of the internal behavior and to identify possible improvements. A simple way is to look at the raw activations of the neurons. This is done by selecting an arbitrary neuron, propagating several samples through the network and look at the inputs that activate this neuron the most. However, these results are usually hard to interpret. A better idea is to look at the neurons weights, instead of the activations. Figure 2.13 shows the raw weights of the 96 filters in the first convolutional layer of Alexnet. In a well trained network, these filters are similar to gabor filters, detecting edges or blobs in the input image. Noisy structures within these filters would be indicators for overfitting or an uncompleted training process. The problem is, that these visualizations only make sense for the first convolutional. For subsequent layers, these filters are not that interpretable. Another way is to use the output of the penultimate layer as a code for the input image. To visualize these feature space, a variety of input samples are evaluated and the resulting codes are transformed into a two-dimensional representation using t-SNE [58]. In this visualization, two images are nearby if their codes generated by the network are close to each other. Besides these methods, deconvolution approaches try to project the neuron activations back to the input pixel space [51, 67].

2.3 Summary

In this Chapter we introduced the principles of artificial neural networks. We recapitulated the history of this machine learning approach, outlined the loose relation to biological neural systems and explained the basic concepts. The idea is to solve a complex problem by dividing it into simpler problems. Simple computational units are composed in a multilayer architecture and interlinked by a system of connections to learn a complex function. The training of the network, which simply speaking means tuning of the connection weights, is done by minimizing a loss function, representing a measure of the error between the network output and a target. Layer-by-layer backpropagation of this error in combination with gradient descent algorithms is used to iteratively adapt the weights.

In the second part we introduced Convolutional Neural Networks (*CNNs*), and showed how concepts like the *local connectivity* of neurons or *shared weights* are used to take advantage of the two-dimensional structure of an input image. Finally, we discussed recently proposed network architectures and the ideas behind these approaches. Next, we introduce Generative Adversarial Networks (*GANs*) which are based on the concepts described in this Chapter.

3 Generative Adversarial Networks

Contents

3.1 Motivation

As shown in Chapter 1, generative models can be used for a variety of computer vision applications. However, there are also many different ways to formulate and train these models. The dominant approaches to train generative models so far, are maximum likelihood based methods and related strategies, see Section 1.3. The challenges using these models are difficult approximations of many intractable computations (partition term) and a complex sampling procedure. Therefore, their ability to create large and realistic images is limited.

Ian Goodfellow *et al.* [14] introduced with Generative Adversarial Networks (GANs) a method to train a generative model based on artificial neural networks that sidesteps these problems. Instead of finding a formulation for $P(\mathbf{x})$, a *GAN* incrementally learns to sample directly. This is realized by formulating an adversarial game between two networks. The first network, called generator G, takes random noise as input and creates new data samples. Its counterpart, the discriminator D, tries to distinguish between real data from the training set and the synthesized samples from G. The generators goal is to maximally confuse the discriminator. In this competition, both networks improve their methods until the fake data is indistinguishable for the discriminator. This means that the generator is

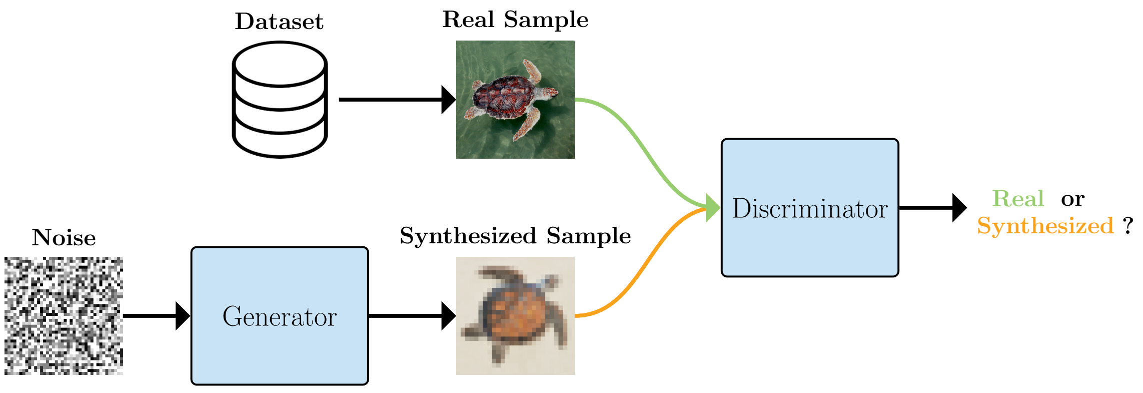


Figure 3.1: Generative adversarial networks: The generator G, takes random noise as input and synthesizes new data samples. Goal of the discriminator D is to distinguish these samples from real data. The generator on the other hand, is trained to make this differentiation as hard as possible.

able to produce new data that is similar distributed as the original data. An illustration of *GANs* is shown in Figure 3.1.

The advantage of *GANs* is, that the entire model is based on artificial neural networks. It can be trained using well studied techniques like backpropagation and stochastic gradient descent. No difficult gradient approximations or sampling procedures are necessary during the training process, like in other generative models. Furthermore, achievements in recently very popular discriminative convolutional networks are applicable for *GANs* as well. For instance, knowledge about efficient network architectures, regularization- or optimization methods can also be used to design and train *GANs*.

3.2 Approach

The aim of *GANs* is to train a generative neural network such that it replicates the empirical data distribution $P_{data}(\mathbf{x})$ of a given dataset as good as possible. For instance, $\mathbf{x}$ could be a color image with $\mathbf{x} \in \mathbb{R}^{W \times H \times C}$ and $P_{data}(\mathbf{x})$ the distribution of car images. As already mentioned the framework to train such a generative network consists of two parts:

Generator G

The generator $G(\mathbf{z}) \rightarrow \mathbf{x}$ is a differentiable function, realized as multilayer neural network. It takes as input a noise vector $\mathbf{z}$ and outputs an image $\mathbf{x}$. The noise values of $\mathbf{z}$ are sampled from a distribution $P_{noise}(\mathbf{z})$. The dimension of the noise vector, which is usually normally or uniformly distributed, is a hyperparameter. Implicitly, the generator defines a distribution $P_G(\mathbf{x})$ over the output samples.

Discriminator D

The second neural network, the discriminator $D(\mathbf{x}) \rightarrow [0, 1]$, is a discriminative model, as the name suggests. It takes an image $\mathbf{x}$ as input and learns to distinguish real from synthesized samples. The scalar output is the probability that $\mathbf{x}$ came from the training set (P_{data}), rather than from the generator (P_G).

To achieve that $P_G \sim P_{data}$, the two networks are pitted against one another in a min-max game given by the objective

$$\min_G \max_D \quad \mathbb{E}_{\mathbf{x}\sim P_{data}}\Big[\log D\big(\mathbf{x}\big)\Big] \;+\; \mathbb{E}_{\mathbf{z}\sim P_{noise}}\Big[\log\big(1 - D(G(\mathbf{z}))\big)\Big], \tag{3.1}$$

where $\mathbb{E}_{\mathbf{x}\sim P_{data}}$ denotes the expectation for samples with distribution P_{data}. Regarding the discriminator, this means that the output probability should be maximized for samples from the dataset and minimized for synthesized data from G. At the same time, the generators objective is to maximize the probability that the discriminator assigns to its samples.

To train both networks using gradient based optimization methods, Equation 3.1 can be rephrased as cost functions. For mini batch gradient descent, with batch size N, the cost function for the discriminator is given as

$$E_D = -\frac{1}{N}\sum_{i=1}^{N}\Big[\log D\big(\mathbf{x}_i\big) \;+\; \log\big(1 - D(G(\mathbf{z}_i))\big)\Big], \tag{3.2}$$

where $\{\mathbf{x}_i, \ldots, \mathbf{x}_N\}$ are samples from the dataset, distributed with P_{data}, and $\{\mathbf{z}_i, \ldots, \mathbf{z}_N\}$ are noise vectors drawn from distribution P_{noise}. This equation is also known as logistic cross entropy loss. Similarly, the cost function for the generator is given as

$$E_G = \frac{1}{N}\sum_{i=1}^{N}\log\big(1 - D(G(\mathbf{z}_i))\big). \tag{3.3}$$

In practice, minimizing $\log(1 - D(G(\mathbf{z}_i)))$ does not provide useful gradients, especially in the beginning of the learning process, when D is confident in detecting the synthesized samples. Instead, the generator can be trained to maximize $\log(D(G(\mathbf{z}_i)))$. This means, that the probability that D makes a mistake is increased, instead of decreasing the probability that D makes the correct distinction. The corresponding cost function is given as

$$E_G = -\frac{1}{N}\sum_{i=1}^{N}\log\big(D(G(\mathbf{z}_i))\big). \tag{3.4}$$

During the training process, the discriminator and the generator are optimized alternately. Optimizing the networks always to completion is ineffective and leads to overfitting. In practice, k steps are used to train the discriminator, followed by one optimization step for the generator. The complete learning process is summarized in Algorithm 1.

Algorithm 1: Generative adversarial networks training procedure using mini batch gradient descent.

```
randomly initialize the parameters of discriminator ($w_D$) and generator ($w_G$)
for number of learning iterations do
    for $k$ steps do
        - Get $N$ data samples $\{\mathbf{x}_i, \dots, \mathbf{x}_N\}$ from the training set.
        - Sample $N$ noise vectors $\{\mathbf{z}_i, \dots, \mathbf{z}_N\}$ from distribution $P_{noise}$.
        - Update the discriminator weights using the gradient:
              $\nabla_{w_D} \left( -\frac{1}{N} \sum_{i=1}^{N} \left[ \log D\big(\mathbf{x}_i\big) \; + \; \log \big(1 - D(G(\mathbf{z}_i))\big) \right] \right)$
    end
    - Sample $N$ noise vectors $\{\mathbf{z}_i, \dots, \mathbf{z}_N\}$ from distribution $P_{noise}$.
    - Update the generator weights using the gradient:
          $\nabla_{w_G} \left( -\frac{1}{N} \sum_{i=1}^{N} \log \big(D(G(\mathbf{z}_i))\big) \right)$
end
```

This training procedure is repeated until the discriminator is maximally confused and unable to differentiate between P_{data} and P_G. Goodfellow *et al.* [14] showed that the optimal discriminator for a fixed generator is given as

$$D^* = \frac{P_{data}(\mathbf{x})}{P_{data}(\mathbf{x}) + P_G(\mathbf{x})}. \tag{3.5}$$

They also showed that the global optimum of this min-max game is reached when $P_{data} = P_G$. Hence, for an optimally trained generator, the output of the discriminator is always $1/2$. Therefore, the training process can be stopped when the mean output of D converges to this value.

3.3 Network Architectures

As already mentioned, *GANs* consists of two networks, a generative and a discriminative network. In a discriminative neural network, the data representations are contracted from the input to the output layer. In contrast, a generative network expands rather than contracts the data from layer to layer. Both networks can be realized as conventional fully connected neural networks or as convolutional networks. Usually, generator as well as discriminator use the same type of architecture.

3.3.1 Fully Connected GANs

In fully connected neural networks, the data contraction or expansion can be easily realized by decreasing or increasing the number of neurons from layer to layer. The output of these networks are alway vectors, therefore the generator output must be reshaped to the desired structure. For instance, a 3072 dimensional output vector can be reshaped to a $32 \times 32 \times 3$ RGB color image. An example for a fully connected *GAN* is shown in Figure 3.2. Since the number of weights significantly increases with the number of neurons per layer, this type of network architecture is usually used for simple datasets containing small images. Goodfellow *et al.* [14] trained a fully connected *GAN* on MNIST [29] and CIFAR-10 [27]. Resulting synthesized images are shown in Figure 3.3.

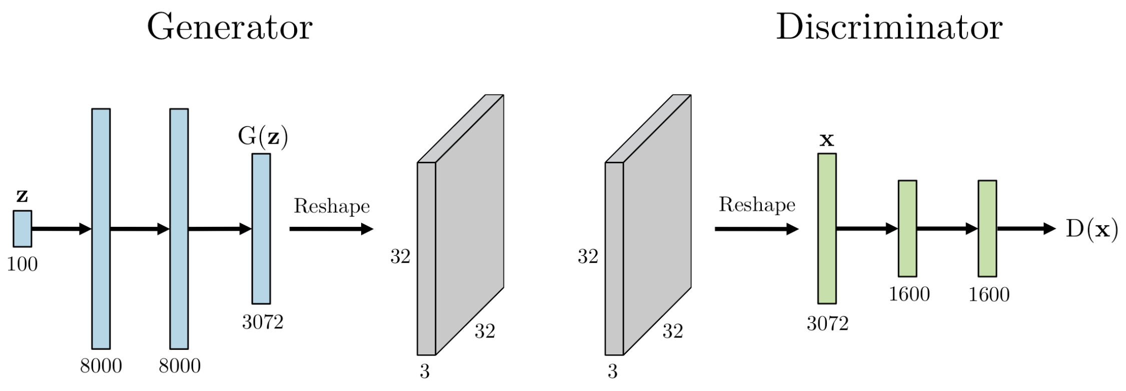


Figure 3.2: Fully connected *GAN* Architecture: In the generator network, the small noise vector $\mathbf{z}$ ix expanded by fully connected layers to an 3072 dimensional output vector. This representation is then reshaped to on $32 \times 32 \times 3$ RGB image. In the discriminator, two fully connected layers are used to determine the probability that the vectorized input image $\mathbf{x}$ is a real sample from the training set.

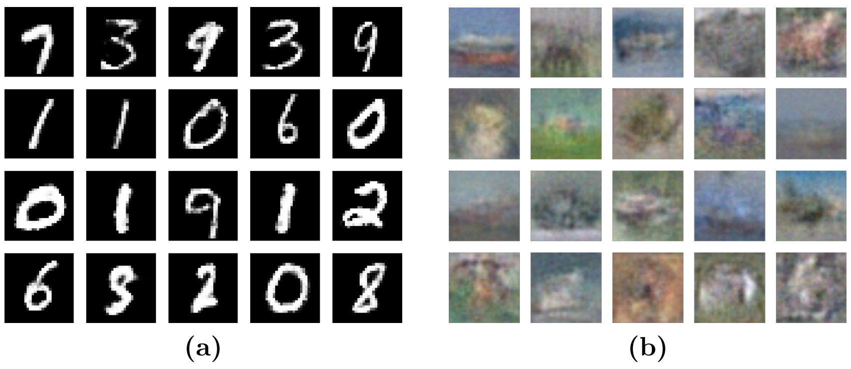


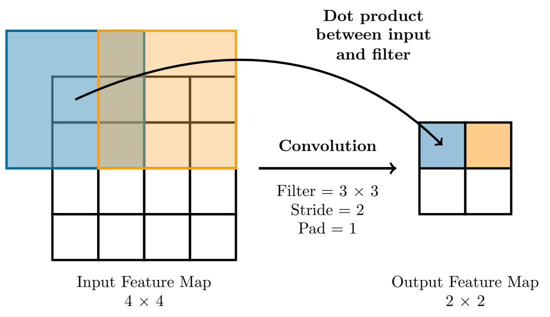
Figure 3.3: Synthesized samples from a fully connected *GAN* trained on MNIST (a) and CIFAR-10 (b). Images taken from [14].

3.3.2 Convolutional GANs

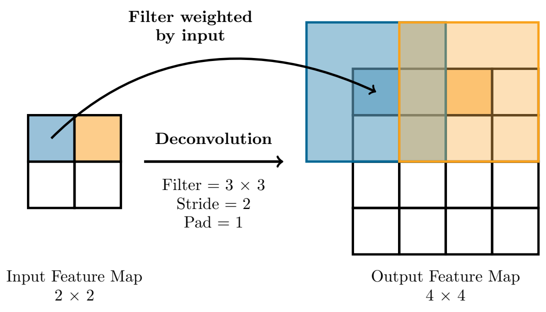
Recently very popular convolutional neural networks are usually discriminative networks. In these architectures, the data dimensionality is decreased with every convolutional layer. From width and height of the input volume, to spatially very small data representations in the last convolutional layer. In generative neural networks it is the other way round. Starting with a small noise vector, the amount of data increases with every layer to the final output image. This upsampling is realized by so called deconvolution layers. Therefore, we first describe this type of layer before we show the concrete architecture of generator and discriminator.

3.3.2.1 Deconvolution Layer

The deconvolution layer [35] implements the inverse operation of a convolution layer. It can be seen as an upsampling layer which expands the feature map in a learnable way. In a convolutional layer with stride S, the filter is virtually placed on the image and a dot product between filter and image is computed, resulting in one element of the output. Now, the filter is S-times shifted along width and height respectively, to compute the next output element. In contrast, in a deconvolution layer the filters are copied to the output instead of computing a dot product. The filters are weighted by the input and then placed on the output, as shown in Figure 3.4. The stride in this case determines the shift in the output where the filters are placed. This means one step in the input corresponds to S steps in the output. In the overlapping areas they are summed up for every position. This operation is equivalent to the backward pass in a convolutional layer. More precisely, the deconvolution forward operation is identical to the backward operation in a convolutional layer and vice versa. Therefore, this layer is also called *backward strided convolution* or *fractional strided convolution*.



(a)

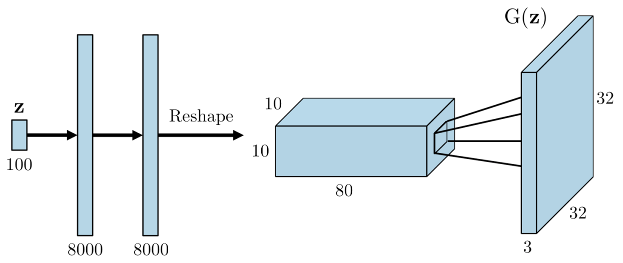


(b)

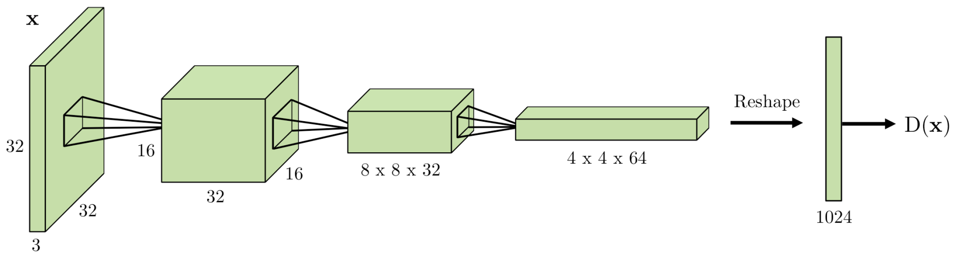
Figure 3.4: The deconvolution layer (b) implements the inverse operation to the convolution layer (a). Instead of contracting the data, the deconvolution layer upsamples the input.

3.3.2.2 Architecture

The convolutional *GAN* architecture proposed in [14] uses a mixture of fully connected, convolutional and deconvolution layers. The generator, shown in Figure 3.5a, firstly uses two fully connected layers to expand the 100 dimensional input noise vector. This vector is then reshaped into a data array and upsampled by a deconvolution layer to the final output image $G(\mathbf{z})$. On the other hand, the discriminator shown in Figure 3.5b, uses three convolutional layers and one fully connected layer. The last layer outputs the probability that the input image $\mathbf{x}$ is a real sample. Synthesized samples from a convolutional *GAN* trained on CIFAR-10 [27] are shown in Figure 3.6.



(a)



(b)

Figure 3.5: Convolutional *GAN* framework: The generator (a) uses a combination of two fully connected layers and one deconvolution layer to create the output image given a noise vector. In the discriminator (b), three blocks consisting of a convolutional and a pooling layer are followed by a single fully connected layer to compute $D(\mathbf{x})$.

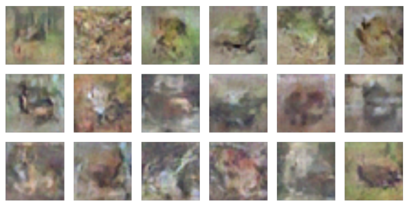


Figure 3.6: Synthesized images from a convolutional *GAN* trained on CIFAR-10. Image taken from [14].

3.4 Extensions

The results shown in Figure 3.3 and Figure 3.6 highlight the potential of the framework to synthesize realistic images. But it can be seen that the *GAN* architectures used in [14] perform very well on simple datasets like MNIST, but tend to produce noisy data on more challenging datasets like CIFAR-10. Another problem of *GANs* is that they are hard to train. The training process is often unstable, resulting in meaningless samples produced by the generator. To avoid this scenario, generator and discriminator must be synchronized well during training. Synchronizing in this context means to keep a balance between the performances of generator and discriminator respectively. For instance, optimizing the generator too frequently can lead to a G network that collapses many values of $\mathbf{z}$ to the same output value $G(\mathbf{z})$. These challenges are addressed by several extensions to the *GAN* framework introduced in the last year. Furthermore, additional parameters are incorporated into the model to influence the generation process. In the following we will describe the most significant *GANs* extensions.

3.4.1 Conditional Generative Networks

Extensions to *GANs* that incorporate additional information are conditional generative adversarial nets [11, 38]. In the unconditioned setting, as described before, there is no possibility to control the generation process. This is addressed by providing contextual data $\mathbf{y}$ to both networks, on which the networks should be conditioned on. Due to this condition, the discriminator expects a specific input distribution and therefore restricts the generator in its output. Thus, it is possible to use the generator in different *modes* and to direct the generative process. For instance, a *GAN* trained on a handwritten digit dataset can be conditioned using the class labels to synthesize a desired digit.

As already mentioned, in a conditional *GAN* the generator $G(\mathbf{z}, \mathbf{y})$ as well as the discriminator $D(\mathbf{x}, \mathbf{y})$ are functions of the contextual data $\mathbf{y}$. The min-max game between generator and discriminator is described by the objective function

$$\min_G \max_D \quad \mathbb{E}_{\mathbf{x},\mathbf{y}\sim P_{data}(\mathbf{x},\mathbf{y})}\Big[\log D\big(\mathbf{x},\mathbf{y}\big)\Big] + \mathbb{E}_{\mathbf{z}\sim P_{noise},\mathbf{y}\sim P_{\mathbf{y}}}\Big[\log\big(1 - D(G(\mathbf{z},\mathbf{y}),\mathbf{y})\big)\Big]. \tag{3.6}$$

The position where the additional information is incorporated is a design decision. For instance, $\mathbf{y}$ could be incorporated only into the input layer or into all layers. In fully connected networks, $\mathbf{y}$ is usually added by a simple vector concatenation. In convolutional networks, the conditional information is integrated as additional channel into the feature maps. An example architecture for a fully connected network is shown in Figure 3.7. Mirza *et al.* [38] trained a conditional *GAN* on the MNIST Dataset [29], where the networks are conditioned on the class labels, encoded as one-hot vectors. Figure 3.8 shows some synthesized samples where each row corresponds to the same condition vector.

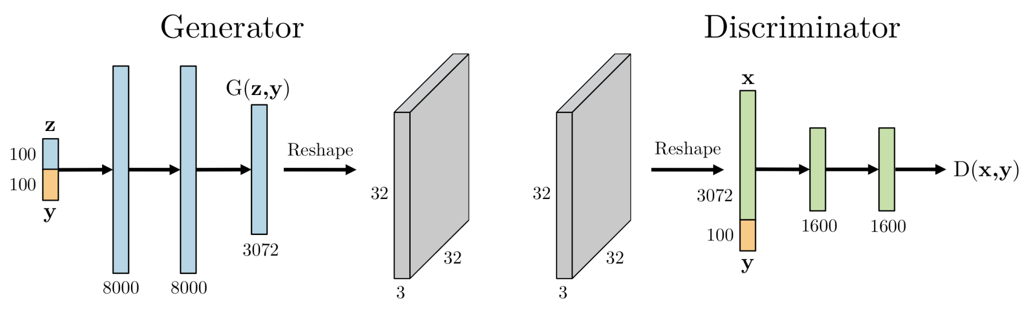


Figure 3.7: Conditional fully connected *GAN*: In this example, the additional information **y** is incorporated into generator and discriminator by a vector concatenation in the input layers.

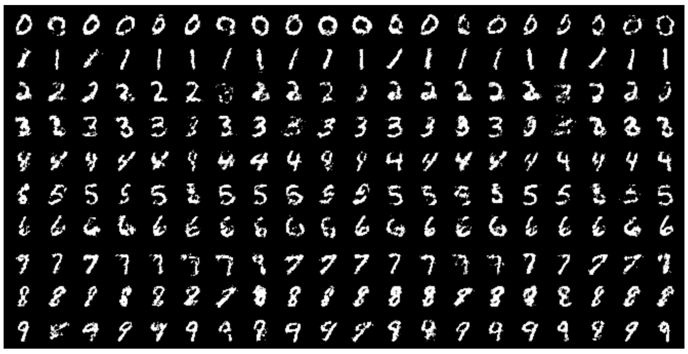


Figure 3.8: Synthesized samples from a conditional *GAN* trained on the MNIST dataset. The class labels, encoded as one-hot vectors, are used as conditional information. Each row corresponds to the same conditional input **y**.

3.4.2 Laplacian Generative Adversarial Networks (LAPGAN)

To improve the image quality, Denton *et al.* [4] exploit the multi scale property of natural images. Instead of a single *GAN*, that has to directly generate the high resolution output, they use a series of *GANs* to break the problem into a sequence of smaller tasks. The idea is that each network predicts the next finer scale and therefore the image is created in a coarse-to-fine fashion. More precisely, each network starts with a coarsened image and predicts only the necessary refinement to the high-resolution image. Each of these networks captures the structures at a particular scale in a Laplacian pyramid. Therefore, they called these framework Laplacian Generative Adversarial Networks (LAPGAN). The framework is based on the conditional *GAN* as described in the last section.

The sampling procedure for a *LAPGAN* is shown in Figure 3.9. Starting with a noise vector $\mathbf{z_3}$ (on the right), the first generator G_3 creates Image $\tilde{I}_3$. This image is upsampled using bilinear interpolation (green) and then used as conditional input (orange) to the next generator G_2. Together with noise $\mathbf{z}_2$, this network creates the difference image $\tilde{h}_2$. By adding this residual image to the coarse input, the fine output image for this scale $\tilde{I}_2$

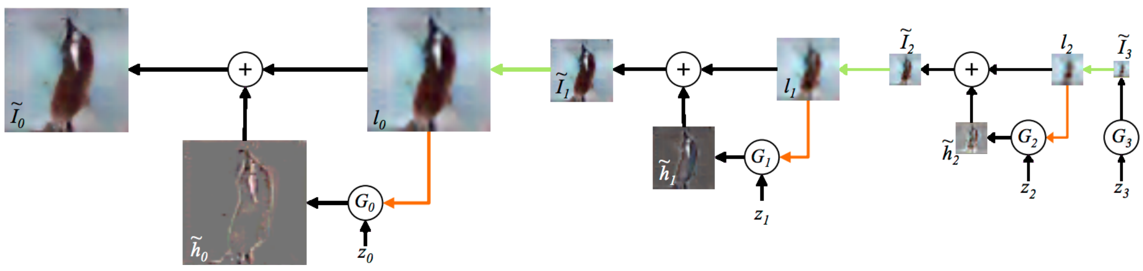


Figure 3.9: *LAPGAN* sampling procedure: Each generator, except the first one, is conditioned on the coarse upscaled output of the previous scale. Note that each generator outputs a difference image, representing the necessary refinements to the fine image. Image taken from [4].

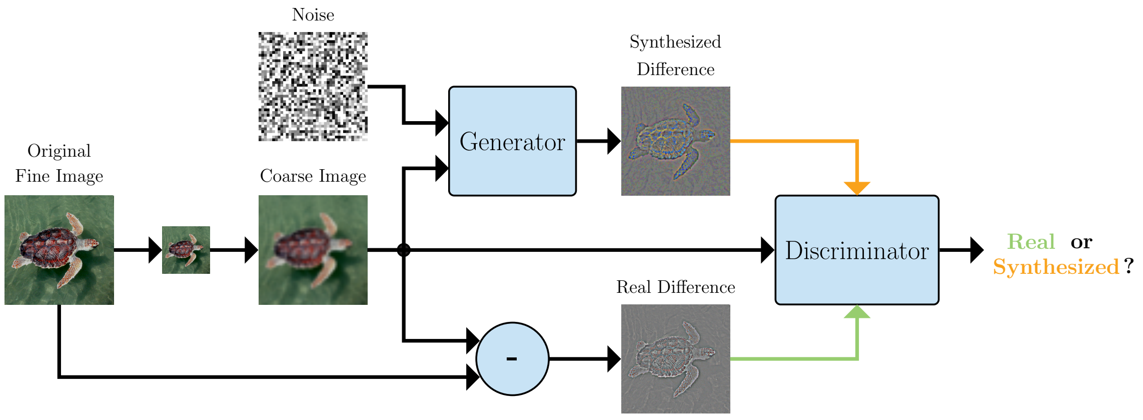


Figure 3.10: *LAPGAN* training procedure for one scale: Both networks are conditioned on the coarse input image. Aim of the discriminator is to distinguish the high-frequency structures synthesized by the generator from a real difference image.

is created. This procedure is repeated until the final full resolution output Image $\tilde{I}_0$ is reached.

The generators at each level within the Laplacian pyramid are learned separately. All generative models are trained using the conditional *GAN* approach, except the first one where no condition is necessary. The training procedure for one scale is shown in Figure 3.10. First, the original fine image is downsampled and subsequently upsampled to create the coarse low frequency image. The inputs to the generator are a noise sample and the coarse image as conditional information. The network outputs the predicted difference to the fine image. Aim of the discriminator, which is also conditioned on the coarse image, is to distinguish between the real and the synthesized difference image. By optimizing Equation 3.6, where $\mathbf{y}$ is the coarse input image, the generator learns to synthesize realistic high frequency structures $G(\mathbf{z}, \mathbf{y})$.

Denton *et al.* [4] demonstrated the performance of their approach on the CIFAR-10 [27] dataset. For the initial scale of 8×8 pixels, they used fully connected layers in combination

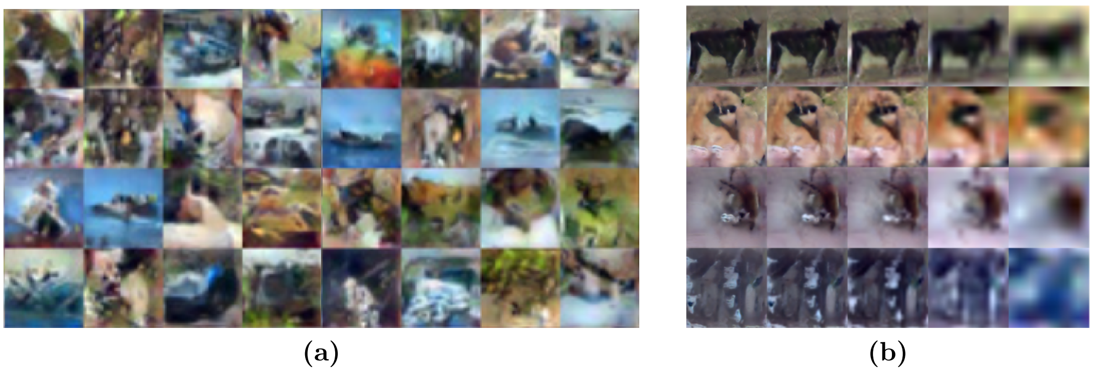


Figure 3.11: *LAPGAN* results: (a) shows samples of a model trained on the CIFAR-10 dataset, (b) visualizes the coarse-to-fine evolution.

with ReLU non-linearities in the generator as well as the discriminator network. To reduce overfitting, dropout is used in the second network. For the subsequent scales, G & D are convolutional networks, where the discriminator uses also fully connected layers in the final stages. Figure 3.11a shows samples synthesized by this *LAPGAN* trained on CIFAR-10. The evolution of synthesized images over different scales is shown in Figure 3.11b. Compared to the standard approach, a *LAPGAN* produces more noiseless and object like images with cleaner defined edges.

3.4.3 Deep Convolutional Generative Adversarial Neural Networks (DCGAN)

A disadvantage of the *LAPGAN* approach is that the learning process is cumbersome and time consuming since each scale must be learned separately. Furthermore, incorporating noise several times within the sampling chain is problematic and leads to unsteady images. This motivated Alec Radford *et al.* [42] to explore different Convolutional Neural Networks (CNN) architectures regarding their applicability for *GANs*. They identified a family of architectures that allow deeper networks and a stable training process even for high resolution images using only a single generator and discriminator network. They called these architectures Deep Convolutional Generative Adversarial Networks (DCGANs).

This class of architectures are characterized by four properties:

Strided Convolution

Motivated by [51] they remove all deterministic spatial pooling layers like maxpooling. Instead, the discriminator uses strided convolutional layers to learn its own spatial downsampling. Similarly, the generator learns the upsampling using backward strided convolution, see Section 3.3.2.1.

Fully Convolutional

Following a recent trend to remove fully connected layers, both networks use only convolutional layers. The highest convolutional layers in discriminator and generator are directly connected to the outputs. This is achieved by setting the kernel size, the stride parameter and the amount of padding such that the output of the generator has the correct image dimensions. Similarly, the discriminator parameters are chosen such that the output is of dimension $1 \times 1 \times 1$, representing the probability of a real input sample.

Batch Normalization

Alec Radford *et al.* identified batch normalization as essential to get deep generator architectures to begin learning. Furthermore, it prevents the generator from mapping all outputs to a single point, which is a common problem in *GANs*. They use batch normalization in both networks, except for the generators output layer and the discriminators input layer.

Activation Functions

In the generator they use Rectified Linear Unit (ReLU) as activation function for all hidden layers and a *tanh* function in the output layer. They argued that a bounded activation function in the last layer allows the network to learn more quickly to saturate and to cover the color space. For the discriminator network, they found that a Leaky Rectified Linear Unit (leaky ReLU) as non-linearity works well, especially when higher resolutions are used.

Figure 3.12 shows an example architecture for an *DCGAN* generator. The 100 dimensional uniformly sampled noise vector is reshaped to a feature map with 100 channels ($1 \times 1 \times 100$) and represents the start of a convolution stack. Each deconvolution layer uses a stride of 2 to double the spatial dimensions until the final output size is reached. In contrast, the number of feature maps is halved from layer to layer. The discriminator architecture is always the exact mirrored version of the generator. Starting with the input image, a stack of convolutional layers with stride 2 is used to reduce the spatial dimensions from layer to layer. The number of feature maps as well as the kernel sizes are equal to the generator. The only difference is that the output of the final layer has only one channel ($1 \times 1 \times 1$) instead of 100 representing the probability of a real input image.

To show the potential of their approach, they trained a *DCGAN* on the LSUN bedroom dataset [66]. This dataset contains about 3 million samples of 64×64 images. The resulting synthesized images are shown in Figure 3.13. This demonstrates that a *DCGAN* is able to generate high quality images for even high resolutions using only a single generator-discriminator pair.

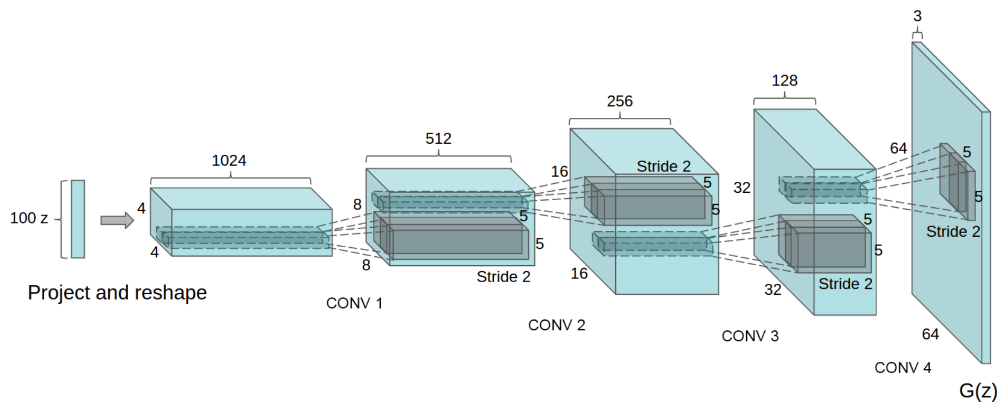


Figure 3.12: Architecture of a *DCGAN* generator: The 100 dimensional noise vector is extended by four deconvolution layers to the final output. Image taken from [42].

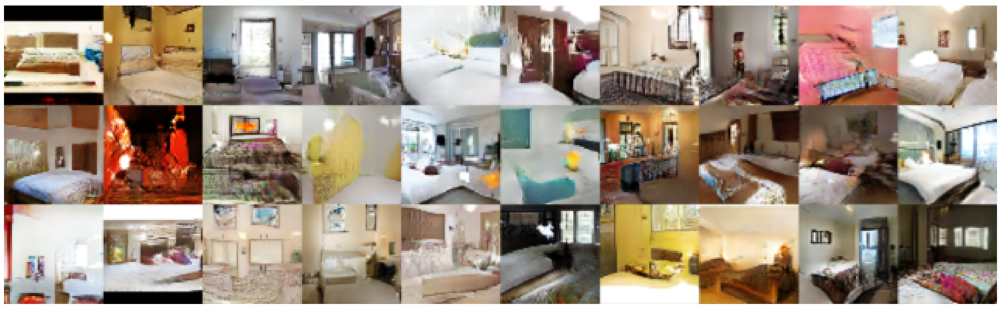


Figure 3.13: Synthesized images from a *DCGAN* trained on the LSUN bedroom dataset. Image taken from [42].

3.5 Summary

In this Chapter we described the principles of Generative Adversarial Networks (*GANs*). Instead of finding a formulation for the data distribution, adversarial nets incrementally learn to synthesize samples that are similar distributed. This is realized by a min-max game between a generator network that creates new samples, and a discriminative network that tries to distinguish between real and synthesized data. The training objective for the generator is to maximally confuse the discriminator by producing *data-like* samples.

We formally described this approach and explained the training procedure where both networks are trained simultaneously. Then, we discussed the originally proposed network architectures and finally, we present recent extensions to *GANs*. On the one hand, these extensions incorporate additional information to the networks to control the generation process, and on the other hand, they adapt the network architecture to stabilize the training process and to increase the image quality. In the next Chapter we investigate the internals of *GANs* and try to manipulate adversarial nets trained on depth-datasets.

4 Exploring Internals of Generative Adversarial Nets

Contents

In this Chapter we investigate properties of Generative Adversarial Networks (GANs) trained on depth datasets. Therefore, we apply them to two common datasets for head- and hand-pose estimation and analyze the resulting generator networks in a variety of ways. On the one hand, we evaluate the quality of synthesized samples and on the other hand, we investigate the internal generation process. This is done by exploring the latent noise space and studying the influences on the resulting output images. Furthermore, we try to manipulate the generator in order to change properties of the synthesized samples. Finally, we train the *GANs* in a conditional setting to predefine the generated pose.

4.1 Datasets & Preprocessing

We use two common datasets to train the *GANs*: The Biwi database and the NYU dataset, containing data of head- & hand-poses, respectively. Note that both datasets provide RGBD-samples, consisting of a RGB image and the corresponding depth-information. In the following experiments we only use the depth-data samples. Details on these datasets and our preprocessing steps are given below.

4.1.1 Biwi Kinect Head Pose Database

The Biwi dataset [9] contains depth data of head poses captured with a Kinect sensor. It consists of roughly 15k frames showing 20 different persons (14 men and 6 woman). The participants were sitting in front of the sensor and were asked to rotate their head around to span all possible yaw and pitch angles. Each frame includes a RGB image (Figure 4.1a), the corresponding depth data (Figure 4.1b) and an annotation. The latter contains the head center location, a binary mask for the face region and the rotation angles. The rotations range around ±75° for yaw, ±60° for pitch and ±50° for roll. The background is removed using a threshold on the distance.

In order to use this data as training set for a *GAN*, we apply the following preprocessing steps. First, we segment the face in the depth data using the provided mask. Then, we cut out a quadratic patch around the provided head center, where the patch size depends on the depth. This guarantees a similar head size within all training samples. Then, each patch is resized to 64 × 64 pixels using nearest-neighbor interpolation. To normalize the data into a range from -1 to 1, we scale the depth values such that all face points are between -1 and 0.5 and the background pixels are equal to 1. To augment the dataset to twice the size, we mirror all samples horizontally. Figure 4.1c shows resulting training samples. To train our *GANs* we use the samples of all 20 persons. It can be seen that several samples are noisy and partially distorted. The reasons are a poor signal-to-noise ratio of the used consumer depth sensor and missing pixels due to occlusions. They arise from the displacement between the sensor and the IR light.

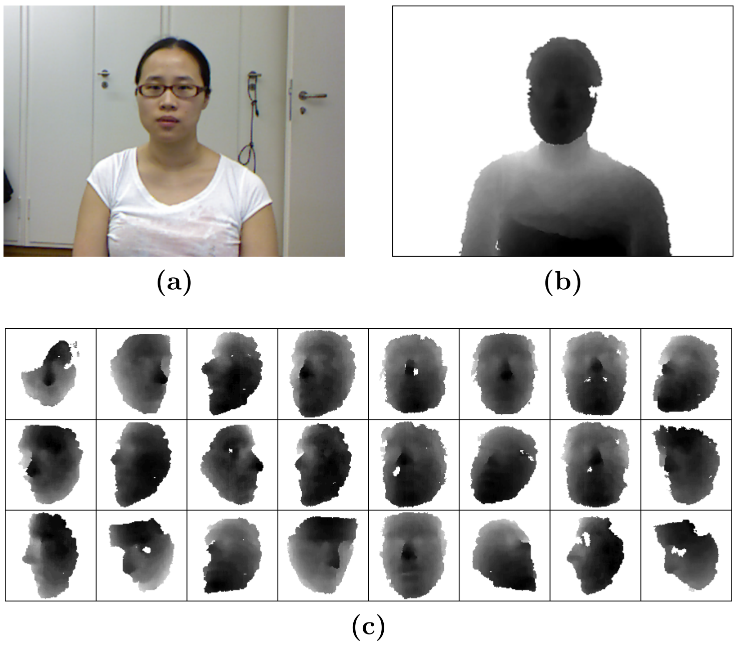


Figure 4.1: Biwi Kinect Head Pose Database: The dataset contains about 15k frames of 20 people recorded with a Kinect depth sensor while turning their heads arround. Each frame consists of a RGB image (a) and the corresponding depth data (b). During the preprocessing, we segment the face region and cut out quadratic patches (c).

4.1.2 NYU Hand Pose Dataset

The NYU Hand Pose Dataset [56] contains RGBD data for hand pose estimation. It includes roughly 80k frames showing different hand poses. For each frame, they provide a RGB image (Figure 4.2a) and the corresponding depth data (Figure 4.2b). Additionally, a synthetic rendering of the hand and the ground truth position of 36 hand locations are available (Figure 4.2c).

As training-set for the *GAN* we use the synthetic data samples where we apply the following pre-processing steps. First, we cut out a quadratic region around the hand center location, where the size depends on the depth to guarantee a similar hand size within the training set. Then, we resize the extracted data to 64×64 pixels using nearest neighbor-interpolation. Finally, we scale the depth values such that the nearest location is equal to -1 and the farthest hand point corresponds to 0.5. The background pixels are set to 1 to achieve clear boundaries. Resulting training samples are shown in Figure 4.2d. To train the unconditional *GAN* we use the 72,757 training samples as well as the 8,252 test samples. In case of the conditional *GAN*, we exclude 500 samples from the test-set for later analyses.

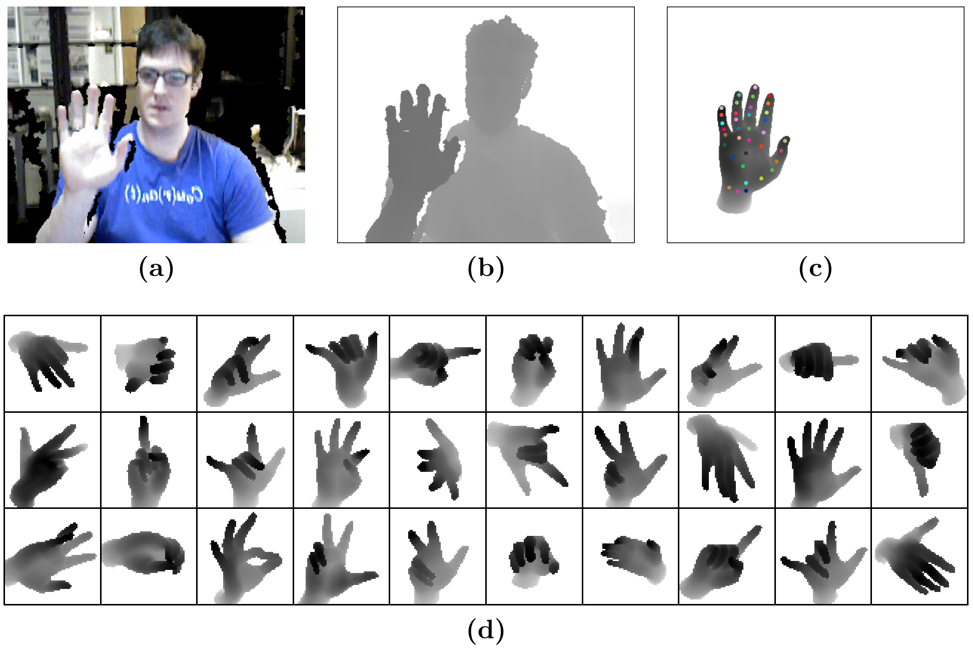


Figure 4.2: The NYU Hand Pose Dataset contains about 80k RGBD samples. In addition to the RGB image (a) and the depth data (b), they provide a synthetic hand rendering as well as positions of 36 key hand locations (c). During the pre-processing we cut-out quadratic patches and scaled them to [-1,1] (d).

4.2 Unconditional GANs

To analyze the performance of adversarial nets on depth data, we train two unconditional *GANs* on the previously described datasets (Biwi & NYU). The network topology is similar to a DCGAN, see Section 3.4.3. We explored a variety of architectures and optimization parameters for each dataset and selected the best performing model regarding the visual quality of the samples. In the following, we will first describe the used models in detail and analyze the resulting samples for each dataset. Then, we will investigate both trained generators in order to obtain a better understanding of the generation process.

4.2.1 Architectures & Results

4.2.1.1 Biwi

The detailed *GAN* architecture for the Biwi dataset is shown in Table 4.1. The generator as well as the discriminator are fully convolutional networks and consist of 5 layers each. The input noise vector **z** is sampled from a uniform distribution [-1,1]. The networks are trained using a mini-batch size of 64 and Adam [26] as optimizer. Samples generated by this model are shown in Figure 4.3. Compared with original data, shown in Figure 4.1c, these samples appear very similar. In most cases, the human face shapes are clearly observable. All possible poses are covered, even though the center position is dominant. The reason is that this pose is also most frequent in the original dataset. The samples have sharp defined edges and small face structures, like noses & eye-holes, are recognizable. Moreover, partial distortions in the original data are also modeled by the network. Although most of the samples contain comprehensible data, about 15% of the samples are destroyed, where no clear head pose is observable. The reason might be the small number of samples for large rotation angles.

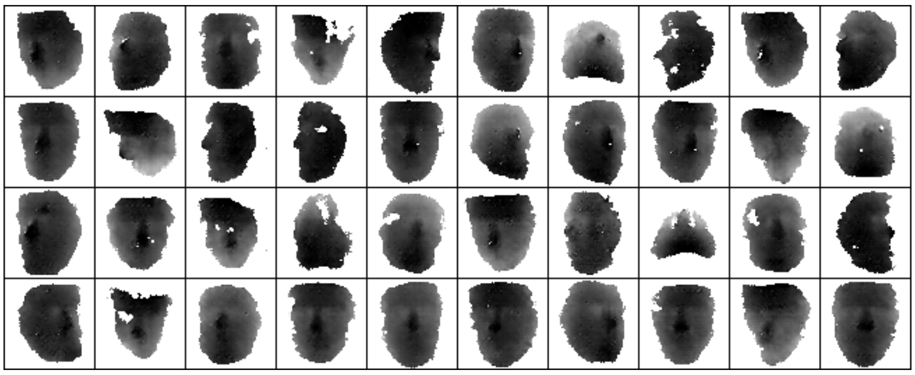


Figure 4.3: Synthesized samples from an unconditional *GAN* trained on the Biwi headpose dataset.

		Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Generator	Type	Deconv.	Deconv.	Deconv.	Deconv.	Deconv.
	Input	$1 \times 1 \times 100$	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$	$32 \times 32 \times 32$
	Parameters:	filter = 256 kernelsize = 4 stride = 1 pad = 0	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 32 kernelsize = 4 stride = 2 pad = 1	filter = 1 kernelsize = 4 stride = 2 pad = 1
	Output:	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$	$32 \times 32 \times 32$	$64 \times 64 \times 1$
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	-
Discrminator	Type	Conv.	Conv.	Conv.	Conv.	Conv.
	Input	$64 \times 64 \times 1$	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$
	Parameters:	filter = 32 kernelsize = 4 stride = 2 pad = 1	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 256 kernelsize = 4 stride = 2 pad = 1	filter = 1 kernelsize = 4 stride = 1 pad = 0
	Output:	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$	$1 \times 1 \times 1$
	Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
	Batchnorm.:	-	✓	✓	✓	-

Training Parameters		optimization algorithm: Adam; learning rate = 0.0003;
		noise distribution: uniform; batch size = 64;				

Table 4.1: Detailed description of the unconditional *GAN* trained on the Biwi headpose dataset.

4.2.1.2 NYU

The unconditional *GAN* trained on the NYU dataset is described in Table 4.2. Similar to the previous *GAN*, generator as well as discriminator consist of five convolutional layers. The difference is that the number of feature maps is twice the size in each layer compared to the Biwi dataset. The reason is that this dataset is much more complex compared to the head pose data, because of the many degrees of freedom a hand has and the finer structures that must be modeled. Figure 4.4 shows samples generated by this model. At first glance, these samples look similar to the original data shown in Figure 4.2d. In the majority, the human hand and the represented pose are recognizable. They have clear defined edges and the depth value patterns are comprehensible. But on closer examination, it is apparent that several samples, about 25-30%, contain invalid data like unrecognizable poses or hands with three fingers. The reason might be the small number of training samples compared to the data complexity.

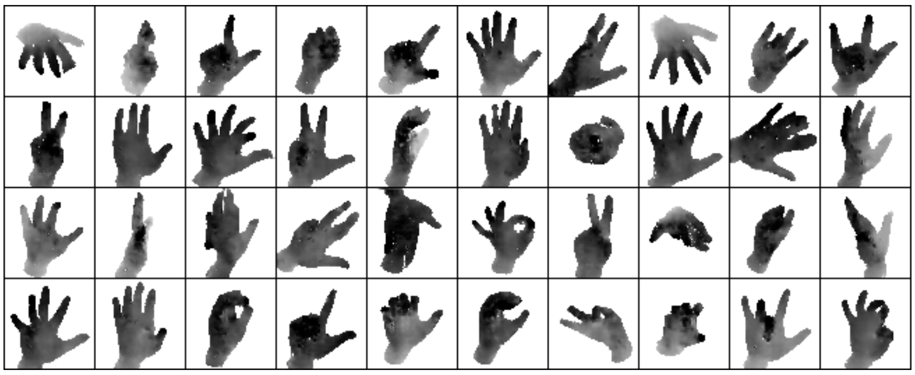
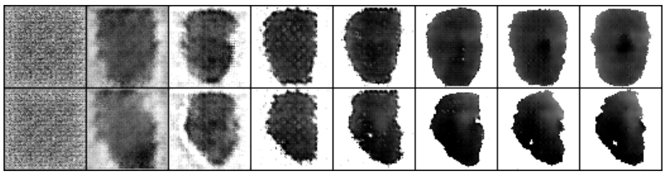


Figure 4.4: Synthesized samples from an unconditional *GAN* trained on the NYU handpose dataset.

		Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
Generator	Type	Deconv.	Deconv.	Deconv.	Deconv.	Deconv.
	Input:	1 × 1 × 100	4 × 4 × 512	8 × 8 × 256	16 × 16 × 128	32 × 32 × 64
	Parameters:	filter = 512 kernelsize = 4 stride = 1 pad = 0	filter = 256 kernelsize = 4 stride = 2 pad = 1	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 1 kernelsize = 4 stride = 2 pad = 1
	Output:	4 × 4 × 512	8 × 8 × 256	16 × 16 × 128	32 × 32 × 64	64 × 64 × 1
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	-
Discrminator	Type	Conv.	Conv.	Conv.	Conv.	Conv.
	Input:	64 × 64 × 1	32 × 32 × 64	16 × 16 × 128	8 × 8 × 256	4 × 4 × 512
	Parameters:	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 256 kernelsize = 4 stride = 2 pad = 1	filter = 512 kernelsize = 4 stride = 2 pad = 1	filter = 1 kernelsize = 4 stride = 1 pad = 0
	Output:	32 × 32 × 64	16 × 16 × 128	8 × 8 × 256	4 × 4 × 512	1 × 1 × 1
	Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
	Batchnorm.:	-	✓	✓	✓	-

Training Parameters		optimization algorithm: Adam; learning rate = 0.0002;
		noise distribution: uniform; batch size = 64;				

Table 4.2: Detailed description of the unconditional *GAN* trained on the NYU dataset.


(a)

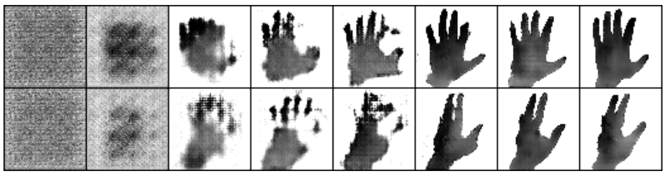

(b)

Figure 4.5: Sample evolution during training: For both datasets, Biwi (a) and NYU (b), we fixed two random noise vectors. During the training, we repeatedly created samples using these starting points. From the first output noise of the untrained network on the left, to the final output on the right.

4.2.2 Analysis

In order to better understand the generation process, we first look at the evolution of synthesized samples during the adversarial game. Therefore, we sample two random noise vectors for each dataset, before we start the learning process. During the training, we periodically use these vectors as input to the generator and save the corresponding output samples. Figure 4.5 shows the resulting sample evolution at noticeable points.

The training process corresponds to the intuitively expected behavior. In the beginning, the generator outputs noise. After a few iterations, the network has learned from

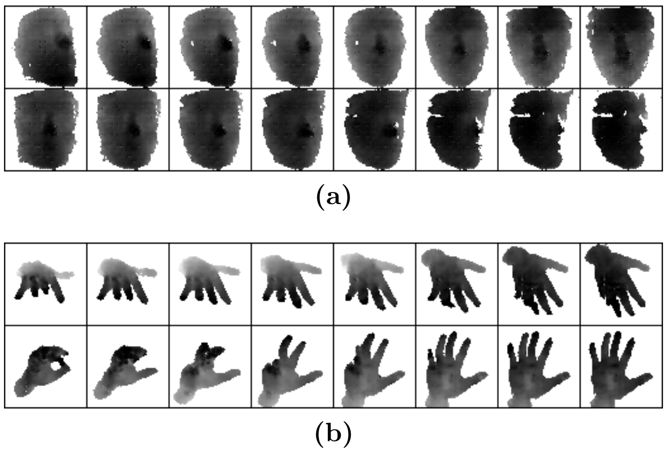


Figure 4.6: Linear interpolation between two random points in the noise space. In each row, we calculated six intermediate points and created the corresponding samples. For both datasets, (a) Biwi and (b) NYU, the transitions between the samples are smooth, indicating that the network did not just memorize existing samples. .

the discriminator that important properties of real samples are the white background and dark centered *blobs*. To mimic this characteristics, the network tries to light up the edge regions. In the next phase, the generator adds structures to confuse its counterpart, even tough they are incoherent or wrongly located. At this point, the discriminator has to focus on more complex properties like correct poses or sharp edges to distinguish real from fake samples. When the generator has learned to model these properties as well, the training process is finished.

Noticeable are mosaic like patterns in early stages of the training process. The reasons are the used deconvolution layers within the generator. Since they use a kernel size of 4 and a stride of 2 pixels, they overlap significantly. In the beginning, the network has not yet learned to coordinate these kernels properly to realize smooth transitions for these overlapping regions.

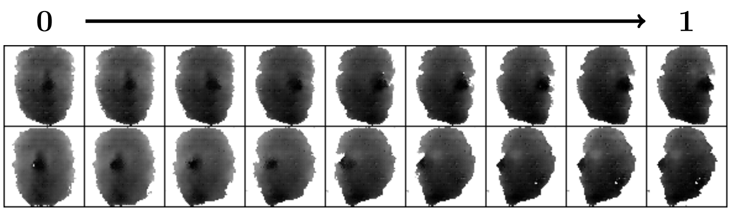
The aim of the next experiment is to better understand the latent noise space. We randomly sample two **z** vectors and linearly interpolate six points in-between. Each row in Figure 4.6 shows the corresponding synthesized depth data. It can be seen that the transitions between the samples are smooth, indicating that the generator did not just memorize samples from the training set. If this would have been the case, the transitions would be sharp and discontinuous.

4.2.3 Manipulating the Generator

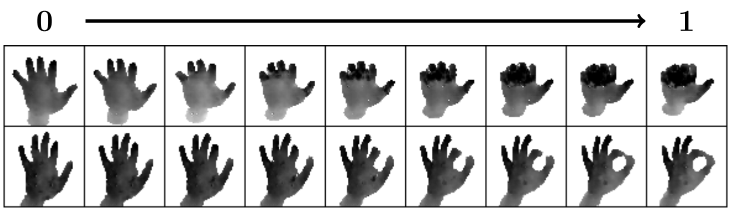
Now we try to control the generator in order to synthesize data with desired properties. For instance, to create depth data of persons looking into a specific direction or to rotate a given sample. The idea is to explore the latent noise space, investigate the impacts on the resulting output data and use this findings to manipulate the generation process.

4.2.3.1 Enhancing Single Noise Components

First, we evaluate the significance of single values within the noise vector. In order to visualize the impact of the i-th component, we use the following approach: First, we randomly sample noise vectors in a small range [-0.1,0.1]. Then, we set the i-th position within the $\mathbf{z}$ vectors to zero and pass them through the generator. Now, we increase the i-th values linearly to 1 and analyze the transformations on the output data. In summary, we found that enhancing single values within the noise vector did not lead to consistent results. It seems that the information for the generation process is rather encoded in combinations of multiple values than in individual components. Figure 4.7 shows examples for the variable influences of single values within the noise vector. In Figure 4.7a the impact of the third $\mathbf{z}$ component to the output of the *GAN* trained on the Biwi dataset is illustrated. Although the initial outputs on the leftmost column are very similar, an increase of this single value has different effects. In the first case the head is rotated to the right, while in the second case the opposite rotation is performed. A similar example for the *GAN* trained on the NYU dataset is shown in Figure 4.7b. In this case an increase of the fifth component can either lead to a closure of the open hand or that thumb and index finger move toward each other. These examples illustrate that manipulating the generation process by enhancing single noise components is not reasonable.



(a)



(b)

Figure 4.7: Examples for the variable influences of single noise components to the output: In case of the *GAN* trained on the Biwi dataset, an increase of the third $\mathbf{z}$ value leads to different head rotations (a). A similar example exists for the *GAN* trained on the NYU dataset (b): Although the initial points are similar, increasing the fifth noise component leads towards different poses.

4.2.3.2 Vector Arithmetic

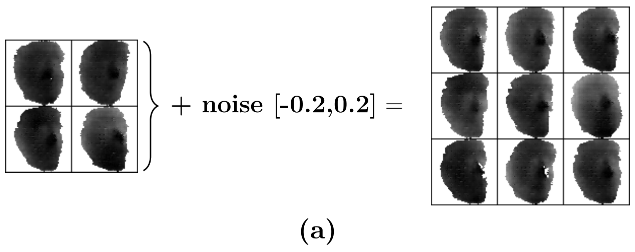
In this Section we investigate the applicability of arithmetic vector operations in the latent noise space, to identify and manipulate properties encoded over multiple **z** positions. Alec Radford *et al.* [42] showed that these operations can be used to influence a *GAN* trained on human faces in order to add or remove attributes. Similarly, we want to use this vector arithmetic to control the pose modeled by the generator.

In the first experiment we try to synthesize depth data of persons in a similar pose. We generate several samples, select four noise vectors where the corresponding outputs show poses directed to the right side and average them. Finally, we use this mean vector as base direction and add uniform noise, ranging from -0.2 to 0.2, to produce new samples. As shown in Figure 4.8a, the head poses of these new samples are also orientated to the right side, although they contain varying head shapes. This example suggests that it is possible to force desired properties in the output by adding specific directions to the input vector.

Now, we try to extend this approach and use arithmetic operations on multiple directions within the noise space to combine properties. The aim is to synthesize partially distorted depth-data of poses directed to the left side. Therefore, we average the noise vectors of four undistorted left-orientated samples and add the mean vector of distorted centered data. To suppress direction changes, without losing the distortion property, we subtract the average of four undistorted centered poses. As shown in Figure 4.8b, this results in consistent and stable generations of left-orientated and partially distorted data.

The aim of the last experiment is to transform an existing pose without loosing properties like the head shape. For the *GAN* trained on the Biwi dataset, we compute a turn vector by subtracting the mean noise vector of four right-orientated samples from the mean vector of four poses directed to the left side. By adding linearly increased portions of this vector to existing examples, we are reliably able to transform their pose to the left side without modifying their head shapes, as shown in Figure 4.9a. Remarkable is that the rotation is linearly modeled in the latent noise space. In the same way, we compute a transformation for the *GAN* trained on the hand pose dataset. In this case, we create a vector from four averaged depth-data samples of open vs. closed hands. By adding linearly increased portions of this vector to existing samples showing closed poses, we are able to linearly open the hand, as shown in Figure 4.9b.

In summary, these examples demonstrate that averaging samples with similar characteristics leads to directions within the noise space that represent the particular property. Combining these directions using vector arithmetic is a reliable approach to generate samples with desired properties, even though these networks are trained in an unsupervised manner.



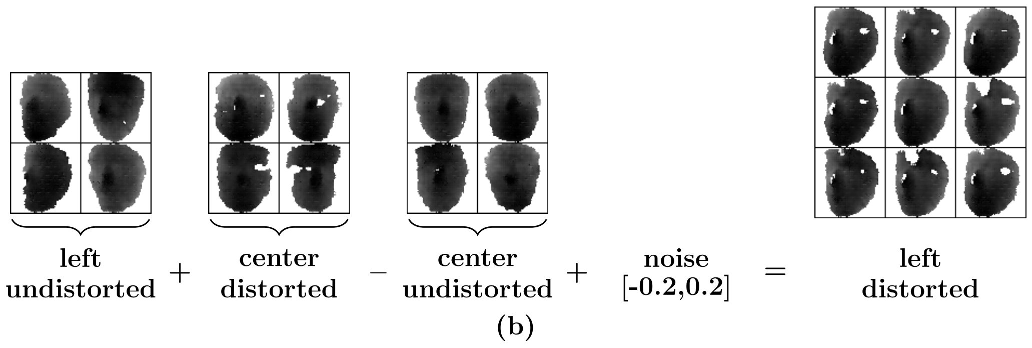
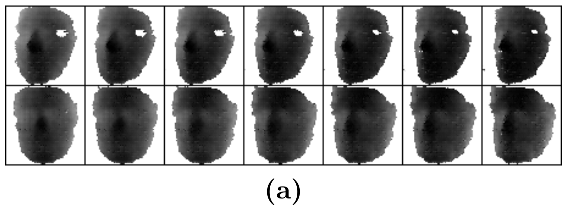


Figure 4.8: Noise space vector arithmetic: (a) By averaging four right-orientated poses and adding small uniform noise, we are able to generate new samples in a similar pose but with varying face structures. (b) By joining several mean vectors, representing different characteristics, we are reliable able to synthesize new samples with combined properties.



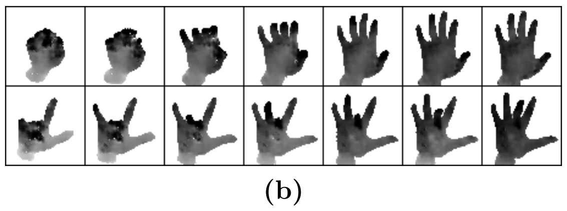


Figure 4.9: Pose transformation: (a) By subtracting the noise vectors of right-orientated samples from noise vectors of left-directed poses, we compute a turn vector. Increasingly adding this vector to existing samples leads to a linear transformation of their pose without a modification of their head shape. (b) The same approach applied to the *GAN* trained on the NYU dataset to transform closed to opened poses.

4.3 Conditional GANs

Both depth datasets additionally provide ground-truth pose information for each frame. In case of the Biwi head pose database, this information contain the pitch, yaw and roll angle of the current pose. The NYU dataset on the other hand, provides the u,v position and the depth of 36 hand locations. In this Section we use these additional data to train conditional *GANs* on both datasets. Thus, it should be possible to determine the synthesized pose by passing the corresponding parameters to the generator.

4.3.1 Biwi Head Pose Database

As already mentioned, the provided ground truth information for this dataset are the head rotation angles. These values range between ±75° for yaw, ±60° for pitch and ±50° for roll. To use them as input, we scale each angle to -1 to 1. In a conditional *GAN*, described in Section 3.4.1, the additional contextual information must be provided to both networks. In case of the generator, the three pose angles can be added to the noise input using a simple vector concatenation. On the other hand, the inputs to the discriminator are two dimensional data. Therefore, we first have to replicate each of these angles to the dimensions of the input (64 × 64), and then add them as additional channels to the input, as shown in Figure 4.10. The network architecture, described in Table 4.3, is similar to the unconditional network except the first layers. We found that reducing the noise dimension to 30 leads to more stable results.

Resulting synthesized samples are shown in Figure 4.11a. By providing the desired angles to the network, we are reliably able to determine the synthesized pose. Similar to the unconditional *GAN*, there was a decline in the sample quality for large angles. The reason is that for these areas the number of samples is significantly smaller compared to the center. As expected, since the whole pose information is now excluded from the noise vector, variations in the latent **z** space do not influence the head pose, as shown in Figure 4.11b.

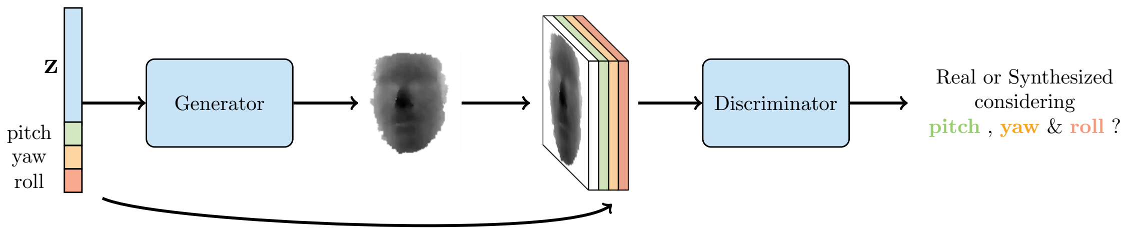
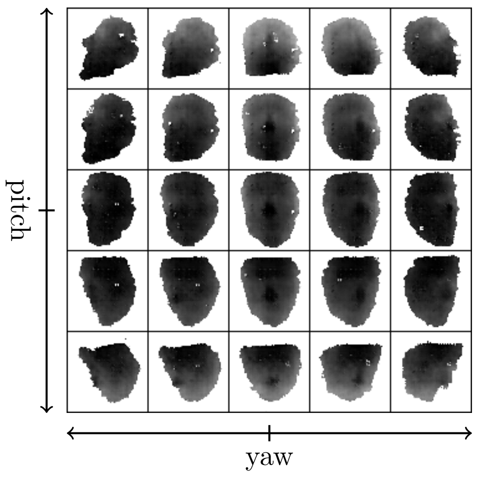


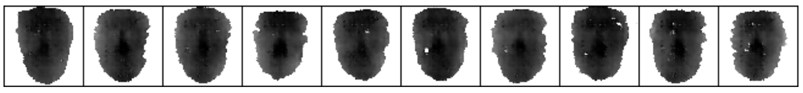
Figure 4.10: Conditional *GAN*: The desired pose parameters are provided to the generator as extension of the noise vector. In case of the discriminator, we incorporate them as additional input channels, where each condition value is replicated to the spatial dimensions of the input.

		Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Generator	Type	Deconv.	Deconv.	Deconv.	Deconv.	Deconv.
	Input	$1 \times 1 \times 33$	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$	$32 \times 32 \times 32$
	Parameters:	filter = 256	filter = 128	filter = 64	filter = 32	filter = 1
		kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
		stride = 1	stride = 2	stride = 2	stride = 2	stride = 2
		pad = 0	pad = 1	pad = 1	pad = 1	pad = 1
	Output:	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$	$32 \times 32 \times 32$	$64 \times 64 \times 1$
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	-
Discrminator	Type	Conv.	Conv.	Conv.	Conv.	Conv.
	Input	$64 \times 64 \times 4$	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$
	Parameters:	filter = 32	filter = 64	filter = 128	filter = 256	filter = 1
		kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
		stride = 2	stride = 2	stride = 2	stride = 2	stride = 1
		pad = 1	pad = 1	pad = 1	pad = 1	pad = 0
	Output:	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$	$1 \times 1 \times 1$
	Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
	Batchnorm.:	-	✓	✓	✓	-
Training Parameters		optimization algorithm: Adam; learning rate = 0.0002; noise distribution: uniform; batch size = 64;				

Table 4.3: Detailed description of the conditional *GAN* trained on the Biwi dataset.



(a)



(b)

Figure 4.11: (a) Synthesized samples from a conditional *GAN* trained on the Biwi dataset. By passing the desired angle parameters to the network, we are able to determine the synthesized pose. (b) Noise variations for fixed yaw, pitch & roll angles do not influence the modeled pose.

4.3.2 NYU Hand Pose Dataset

To train a conditional *GAN* on the NYU dataset, we use the ground truth information of seven hand locations: the finger-tips, the hand center and the heel of the hand. Each of these labels consists of three values, including the u,v position within the image and the depth. Therefore, the conditional information that we provide to both networks is 21-dimensional. The reason why we use seven locations instead of all 36 joint locations is to reduce the network complexity due to limited resources. As described in Section 4.1.2, we initially exclude a test set of 500 samples for later analyses. The detailed network architecture is shown in Table 4.4.

To evaluate the trained network performance, we use the test set annotations as input to the network. On the one hand, this ensures that the conditional inputs are new and on the other hand, that the pose information is plausible. In addition it allows an easy comparison between the synthesized samples and the real depth-data, shown in Figure 4.12. The upper samples in each row are the original depth-data, and the lower samples are the generator outputs. It can be seen that the pose modeled by the network is very similar to the ground truth pose. The real samples differ only on fine details, like cleaner defined edges and smoother transitions. This experiment demonstrates that passing contextual pose information to the *GAN* is a reliable way to influence the network output, even for complex datasets.

		Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Generator	Type	Deconv.	Deconv.	Deconv.	Deconv.	Deconv.
	Input:	$1 \times 1 \times 121$	$4 \times 4 \times 512$	$8 \times 8 \times 256$	$16 \times 16 \times 128$	$32 \times 32 \times 64$
	Parameters:	filter = 512	filter = 256	filter = 128	filter = 64	filter = 1
		kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
		stride = 1	stride = 2	stride = 2	stride = 2	stride = 2
		pad = 0	pad = 1	pad = 1	pad = 1	pad = 1
	Output:	$4 \times 4 \times 512$	$8 \times 8 \times 256$	$16 \times 16 \times 128$	$32 \times 32 \times 64$	$64 \times 64 \times 1$
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	-
Discrminator	Type	Conv.	Conv.	Conv.	Conv.	Conv.
	Input:	$64 \times 64 \times 22$	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$
	Parameters:	filter = 64	filter = 128	filter = 256	filter = 512	filter = 1
		kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
		stride = 2	stride = 2	stride = 2	stride = 2	stride = 1
		pad = 1	pad = 1	pad = 1	pad = 1	pad = 0
	Output:	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$	$1 \times 1 \times 1$
	Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
	Batchnorm.:	-	✓	✓	✓	-
Training Parameters		optimization algorithm: Adam; learning rate = 0.0002; noise distribution: uniform; batch size = 64;				

Table 4.4: Detailed description of the conditional *GAN* trained on the NYU dataset.

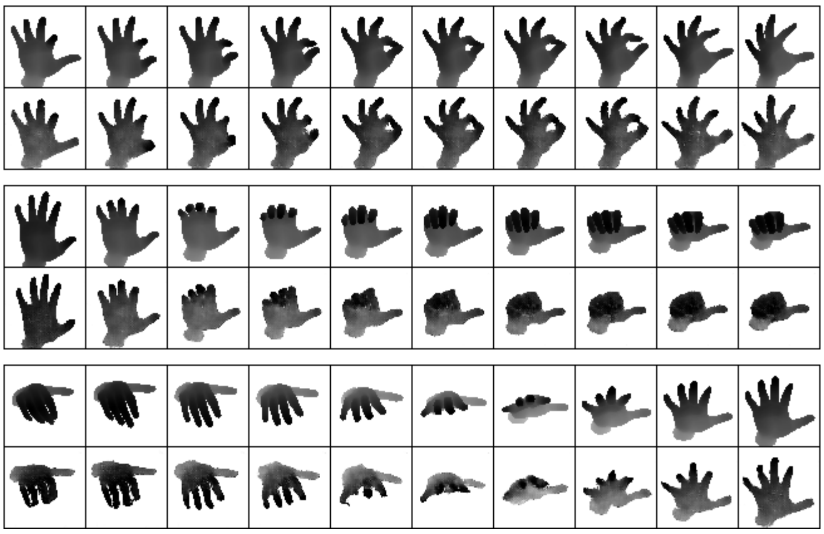


Figure 4.12: Conditional *GAN* trained on the NYU dataset: The upper rows show the original test samples and the lower rows the synthesized data, where the annotations of the test samples are used as conditional input to the generator.

4.4 Summary

In this Chapter Generative Adversarial Networks (*GANs*) were investigated in order to identify ways to manipulate the generation process. We trained unconditional *GANs* on two depth-datasets and demonstrated that they were able to synthesize new realistic samples of human poses. In order to investigate how semantic properties of the synthesized samples are encoded in the initial vector, the latent noise space was explored. We found that relevant information for the generation process is rather encoded in combinations of multiple values than in single components. Then, we show that a reliable way to identify semantic properties within the latent space is to average the noise vectors of synthesized samples sharing the same property. Moreover, by combining these directions using vector arithmetic it was possible to synthesize samples with desired properties. Finally, conditional *GANs* were trained on both datasets, where the pose information was provided to generator and discriminator, allowing a more precise control over the synthesized pose. However, the disadvantage of this method is that data annotations are necessary to train the conditional adversarial nets. In the next Chapter we analyze the applicability of *GANs* to solve two fundamental computer vision tasks.

5 Applications

Contents

The aim of this Chapter is to investigate the applicability of Generative Adversarial Networks (GANs) to solve two fundamental computer vision problems. First, we analyze the potential of *GANs* for unsupervised representation learning by reusing parts of trained discriminator and generator networks. In the second part, we try to use adversarial networks for domain specific image upsampling.

5.1 Unsupervised Feature Learning

Learning feature representations from unlabeled data is an active research area in computer vision. The goal is to reduce the data dimensionality by learning representations that capture latent structures within the high dimensional input without using any data annotations. The extracted features can later be reused to learn the supervised tasks of interest.

One possibility to realize this unsupervised encoding-task is to train an artificial neural network like an autoencoder. Lower layers detect simple features which are then used by subsequent layers to model more complex features. This network can then be reused to initialize parts of a larger network for some supervised task. Another approach is to reuse this network as fixed feature extractor and train a second supervised network on top of it. However, the idea is that most of the work to extract useful features is already done by the initial unsupervised training, and less labeled data is necessary to learn the supervised task.

In the context of computer vision, it is often easy to collect unlabeled images or videos, for instance using web search engines. The problem, though, is that annotations are expensive to collect. Therefore, aim of unsupervised feature learning is to take advantage of these large amounts of unlabeled data. Since *GANs* can be trained without any annotations, they are a candidate for unsupervised feature learning. In this Section we investigate the capabilities of adversarial nets to build good image representations for supervised classification or regression tasks.

5.1.1 Methods

To utilize *GANs* for unsupervised feature learning, we reuse parts of trained generator and discriminator networks as fixed feature extractor or as initialization for the supervised task of interest.

5.1.1.1 Discriminator as Feature-Extractor

Aim of the discriminator network within a *GAN* framework is to distinguish real data from synthesized samples. In the final phase of the training process, the samples from the generator network are very similar to the original data. Therefore, the discriminator has to focus on fine details and extract features that allow this distinction. The idea now is, that these learned image representations could also be useful for other supervised tasks. Therefore, we remove the last layer and use the remaining network as fixed feature extractor. Thus, the dimension of the representation is defined by the output dimensions of the penultimate discriminator layer. In a Deep Convolutional Generative Adversarial Networks (DCGANs) architecture, see Section 3.4.3, the penultimate layer is a convolutional layer which outputs a feature map. This volume can either be flattened to a vector, or directly used as input to a convolutional layer in the subsequent network. However, essential is that the feature extracting network is fixed and will not be changed during the following supervised learning process.

5.1.1.2 Discriminator Fine-Tuning

Another way is to incorporate the trained discriminator into a larger network and fine-tune all layers to solve the supervised problem. Similarly to the previous method, the last layer of the trained discriminator is removed first. Then, depending on the supervised task, new classification or regression layers are added on top of it. The difference to the last approach is that during the supervised learning, the weights of the pretrained network are also updated by continuing the back-propagation. This process is known as parameter *fine-tuning* and illustrated in Figure 5.1. Compared to a fixed feature-extractor, this allows an adaption of the data representations depending on the given task.

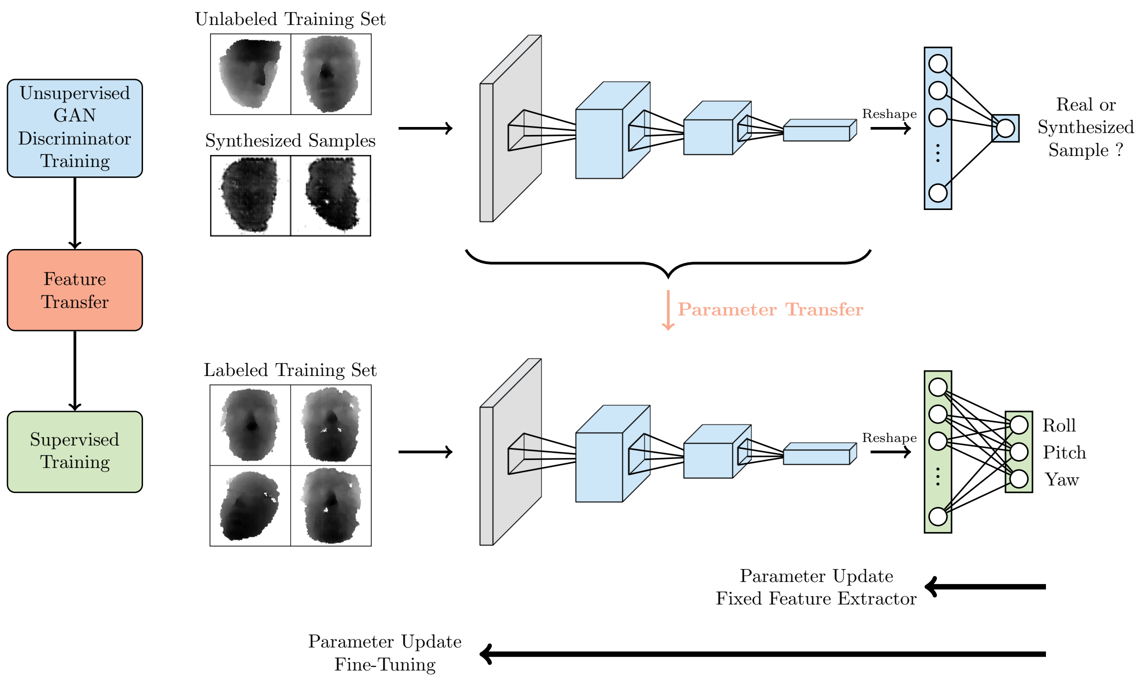


Figure 5.1: Discriminator Parameter Transfer: First, an unconditional *GAN* is trained on a large unlabeled dataset. Then, all layers of the discriminator except the last layer are transferred into a new network, where a new output layer is added for supervised learning. The discriminator part can then either be used as fixed feature extractor or fine-tuned during the supervised training.

5.1.1.3 Inference Network

The idea behind this approach is to exploit the property of a generative model to synthesize huge amounts of new data samples. This is realized as follows: First, an unconditional *GAN* is trained on a large unlabeled dataset. Then, the generator is fixed and a third network, an *inference net*, is trained in order to learn the inverse operation to the generator. This network predicts the input noise vector, given a synthesized sample. Reconstructing this $\mathbf{z}$ vector might lead to good feature representations, since all properties of a generated sample must be encoded within this vector. It can be seen as an autoencoder, where the encoding part is trained after the decoding part. The advantage is that the inference net can be trained on an infinite training set, since the generator can be used to synthesize unlimited samples. During the training process, illustrated in Figure 5.2, we produce on the fly new data samples using the previously learned generator network.

The architecture of this network is the exact inverse to the generator and identical to the discriminator, except the output dimension of the last layer. After completion of the training process, the last layer is removed and the remaining network can be used as fixed feature extractor or as an initialization for supervised tasks, similar to the previous methods.

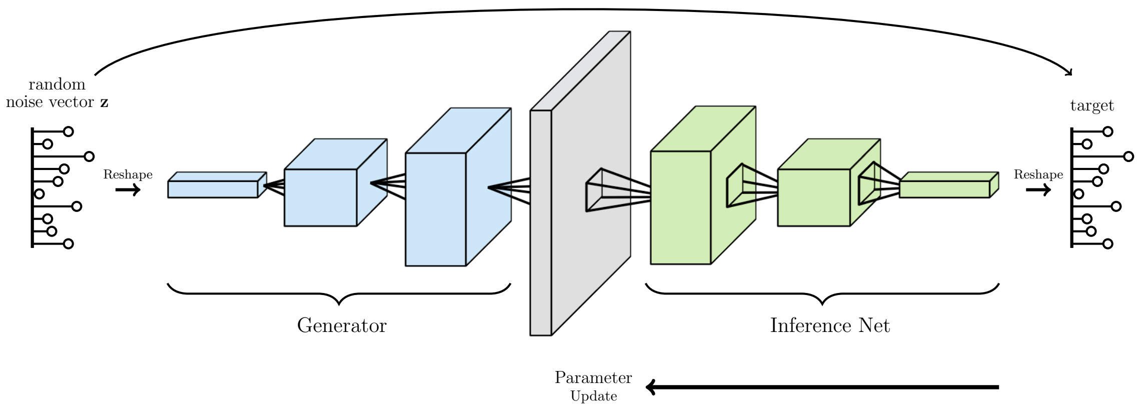


Figure 5.2: Inference Network: After completion of the *GAN* training, we exploit the fixed generator to produce new data samples. These are used to train a subsequent network, an inference net, which tries to reconstruct the input noise vector. The advantage is, that the training set is theoretically infinite. Afterwards, the learned data representations can be reused for supervised tasks.

5.1.2 Classification

To evaluate the capabilities of the previously described unsupervised feature learning methods for classification tasks, we train a *GAN* on the Street View House Numbers (SVHN) dataset [41], and then reuse the learned data representations to train a supervised classification network. The *SVHN* dataset is a real-world database and contains color images of house numbers obtained from a large amount of Google Street View images. In total, the dataset consists of over 600k labeled digits, available in two formats: The first variant contains the original house-number images as they appear in Google Street View with variable resolutions. We use the second format, the cropped digits, where all numbers are resized to 32×32 pixels. Therefore, the bounding boxes are extended to square windows, which may result in images containing multiple numbers. Compared with the handwritten digit classification problem on the MNIST dataset, this problem is significantly harder. The reasons are on the one hand corrupted images due to blur, distortion or illumination effects, and on the other hand, font and style variations. The dataset is divided into *train* ($73,257$ images), *test* ($26,032$ images) and *extra-train* ($531,131$ images) subsets. For each digit they provide a ground-truth label. Figure 5.3a shows samples of this dataset.

We use the digits of the train and the extra-train subset, $604,388$ digits in total, to learn an unsupervised *GAN*. The detailed architecture is described in Table 5.1. The noise vector is uniformly distributed between -1 and 1. As optimization algorithm, we use Adam with a learning rate of 0.0002 and a mini-batch size of 128. We apply no pre-processing to the images except scaling them to $[-1, 1]$. Synthesized samples from this model are shown in Figure 5.3b.

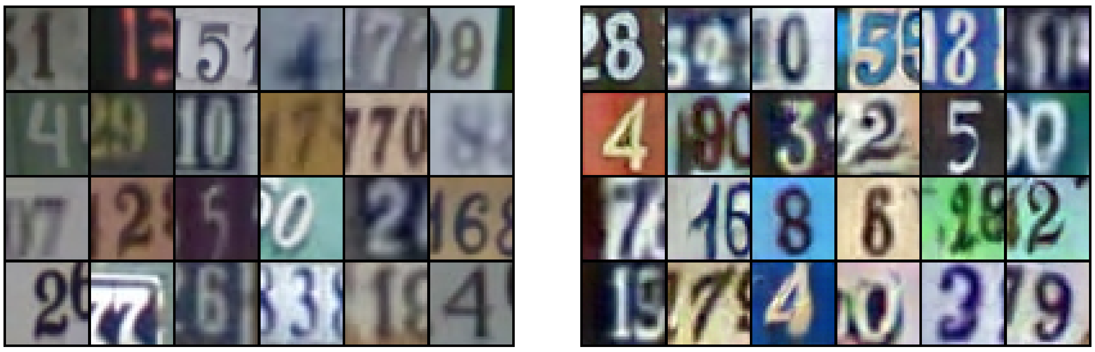


Figure 5.3: *SVHN* dataset: It contains color images of house numbers obtained from Google Street View, resized to 32×32 pixels (a). Synthesized samples from an unconditional *GAN* trained on this dataset are shown in (b).

		Layer 1	Layer 2	Layer 3	Layer 4
Generator	Type	Deconv.	Deconv.	Deconv.	Deconv.
	Input	$1 \times 1 \times 100$	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$
	Parameters:	filter = 256 kernelsize = 4 stride = 1 pad = 0	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 3 kernelsize = 4 stride = 2 pad = 1
	Output:	$4 \times 4 \times 256$	$8 \times 8 \times 128$	$16 \times 16 \times 64$	$32 \times 32 \times 3$
	Nonlinearity:	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	-
Discrminator	Type	Conv.	Conv.	Conv.	Conv.
	Input	$32 \times 32 \times 1$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$
	Parameters:	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 256 kernelsize = 4 stride = 2 pad = 1	filter = 1 kernelsize = 4 stride = 1 pad = 0
	Output:	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$	$1 \times 1 \times 1$
	Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
	Batchnorm.:	-	✓	✓	-

Training Parameters		optimization algorithm: Adam; learning rate = 0.0002;
		noise distribution: uniform; batch size = 128;			

Table 5.1: Detailed description of the unconditional *GAN* trained on the *SVHN* dataset.

After completion of the *GAN* training, we set up a classification net similar to the discriminator. The network architecture for this supervised task, shown in Table 5.2, differs only in the output dimension. Instead of one neuron, indicating a real input sample, the classification network has 10 output neurons representing all digits.

To experimentally compare the unsupervised feature learning methods described in the previous section, we initialize layer 1 to layer 3 with the weights of the trained discriminator and inference net, respectively. On the one hand, we use these layers as fixed feature extractor where the weights are not changed during the training process, and on the other hand, we fine-tune all parameters within the network to solve the classification task. Therefore, we end up with four different networks in total. As

	Layer 1	Layer 2	Layer 3	Layer 4
Type	Conv.	Conv.	Conv.	Conv.
Input	$32 \times 32 \times 1$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$

Parameters:	filter = 64
	kernelsize = 4
	stride = 2
	pad = 1
		filter = 128
		kernelsize = 4
		stride = 2
		pad = 1
			filter = 256
			kernelsize = 4
			stride = 2
			pad = 1
				filter = 10
				kernelsize = 4
				stride = 1
				pad = 0

Output:	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$	$1 \times 1 \times 10$
Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	-
Batchnorm.:	-	✓	✓	-
Training Parameters		optimization algorithm: Adam; learning rate = 0.001; batch size = 100;		

Table 5.2: Detailed description of the classification network trained on the *SVHN* dataset.

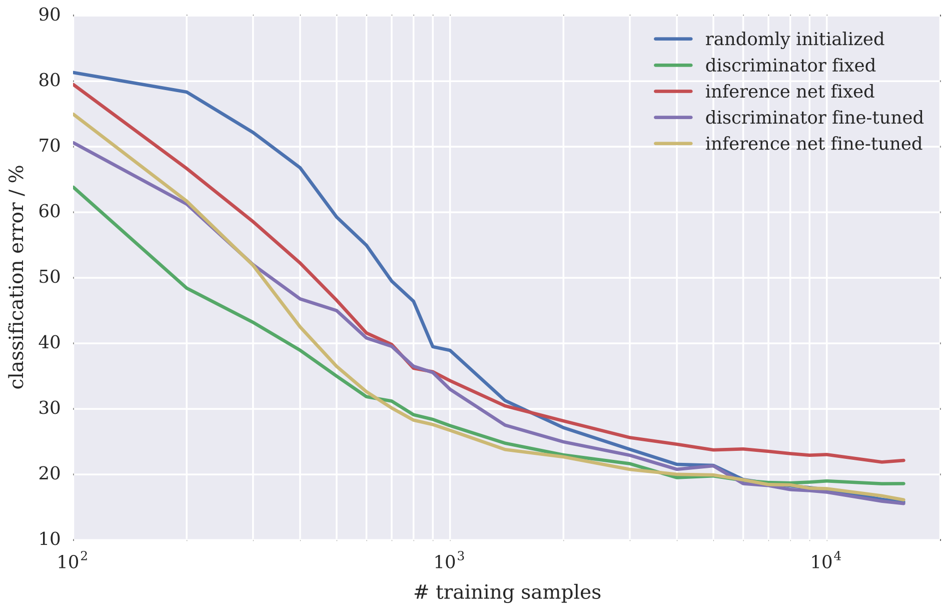


Figure 5.4: Classification error rate on the *SVHN* dataset as a function of the amount of labeled training samples for different unsupervised feature learning methods.

baseline we train a fifth network from scratch, where the weights of all layers are randomly initialized. All networks are trained until convergence, using Adam as optimization algorithm with a learning rate of 0.001 and a mini-batch size of 100. We do this for different amounts of training data, from 100 to 16,000 images, where these subsets are always class uniformly sampled from the original training set. For each trained network, we evaluate the classification performance on the 26,032 test images.

The results of this experimental comparison are shown in Figure 5.4. It can be seen that for small labeled training sets, the classification networks initialized with the weights of discriminator or inference net, perform significantly better than the randomly initialized network. Especially the performance gap to the fixed discriminator network at very small training-sets ($<$ 1000 samples), indicates that the discriminator is able to learn generic features. Only at about 5,000 labeled training samples, the randomly initialized network is able to close the gap to the fine-tuned networks. As expected, for large datasets, the fixed networks perform worse compared to the fine-tuned versions. The reason is that for large amount of training samples, the network is able to learn features beneficial for the particular task. However, a final error rate of 18.6%, based on the fixed discriminator features, indicates that *GANs* are a strong candidate for unsupervised feature learning.

5.1.3 Regression

Now, we analyze the applicability of features learned by a *GAN* for regression tasks. Similar to the last section, we first train *GANs* on unlabeled data samples and then reuse parts of the discriminator and generator networks for supervised tasks when labeled samples are scarce. For these experiments we use the pose estimation depth datasets from Chapter 4.

5.1.3.1 Head Pose Estimation

In the first experiment we train a *GAN* on the Biwi dataset, and then reuse the learned features to train a supervised regression network to estimate the head pose. The dataset and the pre-processing steps are described in Section 4.1.1. The *GAN* architecture, as well as the training parameters are similar to the unconditional *GAN* used in Section 4.2.1.1. The only difference is, that this time we split the dataset into a training- and a test-subset. The training set consists of depth data from 18 persons, the samples of the remaining two persons represent the test set. To train the *GAN*, we use only the training subset.

After completion of the *GAN* training, we set up a regression network to estimate the head pose. The network architecture is described in Table 5.3. Layer 1 to layer 4 are identical to the discriminator or inference net, respectively. The output layer consists of three neurons representing the yaw, pitch and roll angle of the head pose. Similar to the classification experiment, we initialize the first four layers with the parameters of the trained discriminator or inference net and use them as fixed feature extractor or for fine-tuning. To analyze the quality of these pretrained features, we train a fifth network with randomly initialized parameters for comparison. We vary the number of training samples from 100 to 27,000 and train the networks until convergence, using Adam as optimization algorithm. For each trained network, we evaluate the performance on the test set by computing the mean angle error, given as

$$E = \|\mathbf{y}_{target} - \mathbf{y}_{predicted}\|_2 \qquad \mathbf{y} = \begin{pmatrix} \text{yaw} \\ \text{pitch} \\ \text{roll} \end{pmatrix}. \tag{5.1}$$

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Type	Conv.	Conv.	Conv.	Conv.	Conv.
Input	$64 \times 64 \times 1$	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$
Parameters:	filter = 32	filter = 64	filter = 128	filter = 256	filter = 3
	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
	stride = 2	stride = 2	stride = 2	stride = 2	stride = 1
	pad = 1	pad = 1	pad = 1	pad = 1	pad = 0
Output:	$32 \times 32 \times 32$	$16 \times 16 \times 64$	$8 \times 8 \times 128$	$4 \times 4 \times 256$	$1 \times 1 \times 3$
Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	-
Batchnorm.:	-	✓	✓	✓	-
Training Parameters		optimization algorithm: Adam; learning rate = 0.0005; batch size = 50;			

Table 5.3: Detailed description of the regression network trained on the Biwi dataset to estimate the head pose.

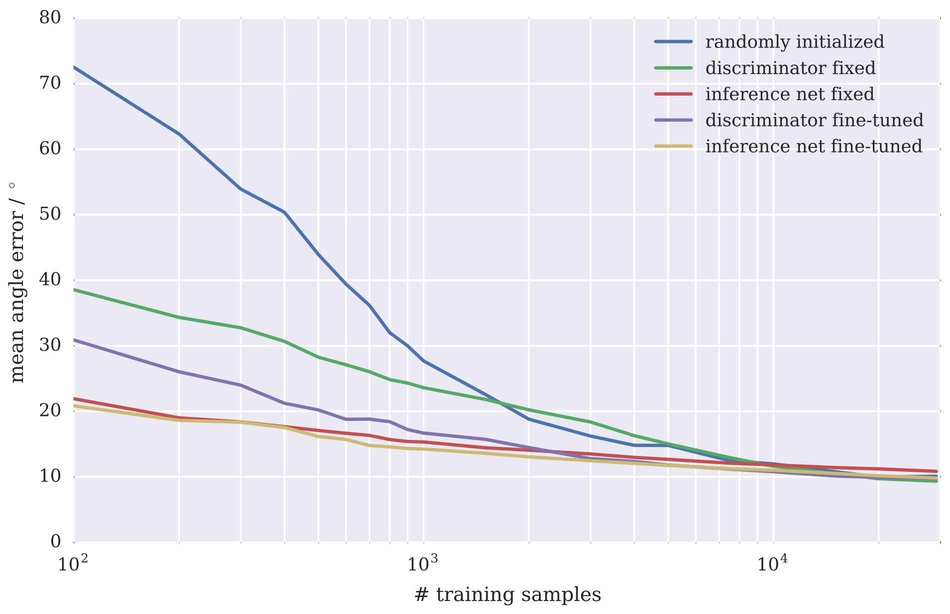


Figure 5.5: Mean angle error on the Biwi dataset as a function of the amount of labeled training samples for different unsupervised feature learning methods.

The results of this experimental comparison are shown in Figure 5.5. The large gap between the randomly initialized network and the networks based on discriminator or inference net for small training sets, demonstrates that *GANs* are able to learn reusable data representations. Especially the features learned by the inference net work very well for this regression task. Using the randomly initialized network, about 15-times more labeled samples are needed to achieve the performance of the model based on the inference net at 100 training samples.

Noticeable is also, that the networks based on the inference net perform better than the discriminator-based models. The reason might be that in the final phase of the *GAN* training, the discriminator focus on fine details that are useful to distinguish the fake samples from real data, but not do determine the pose. Only at about $8,000$ labeled

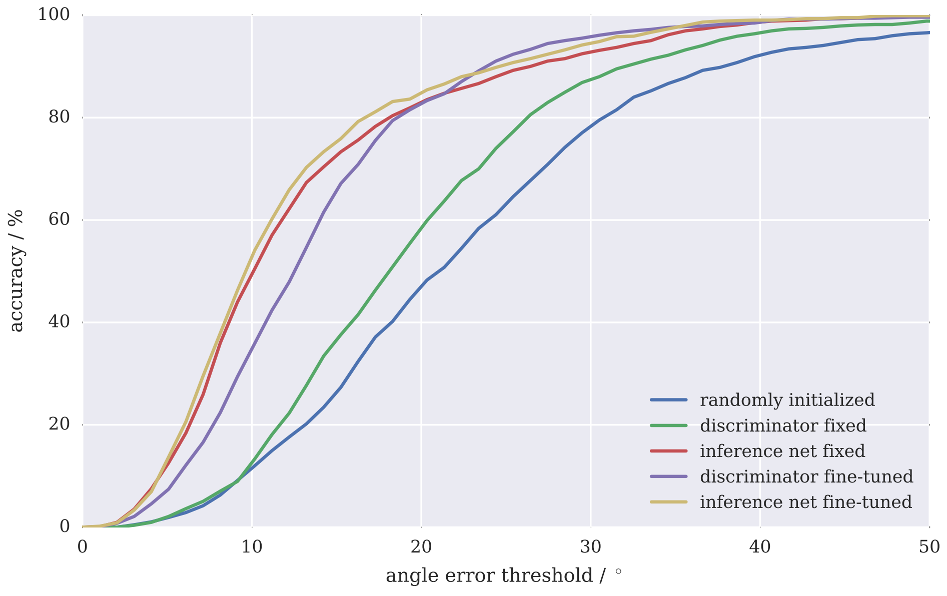


Figure 5.6: Accuracy of the head pose estimation on the Biwi dataset for a training set of 1,000 labeled samples and different unsupervised feature learning methods. The curves show the fraction of correct samples over an increasing success threshold.

training samples, the network trained from scratch is able to close the gap to the pre-initialized networks.

Another common evaluation is to compute the percentage of correctly predicted test samples over an increasing success threshold. Figure 5.6 shows the results of this evaluation for a fixed training-set size of 1000 labeled samples. For a success threshold of 20°, the fine-tuned inference network achieves an accuracy of 85.43%, compared to 48.29% using the randomly initialized network.

5.1.3.2 Hand Pose Estimation

In the second regression experiment, we try to reuse features learned by a *GAN* to estimate hand poses on the NYU dataset. The dataset and the preprocessing steps are described in Section 4.1.2. The network architecture and the training parameters are similar to the unconditional *GAN* described in Section 4.2.1.2. The only difference is, that this time only the samples of the training-subset are used to learn the *GAN*.

Similar to the last experiment, we set up a supervised regression network after completion of the *GAN* training, and reuse the parameters learned by the discriminator or inference net to initialize the network. As pose annotation we use the ground truth information of seven hand locations: the finger-tips, the hand center and the heel of the hand. The reason why we use only these seven locations is to reduce the network

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Type	Conv.	Conv.	Conv.	Conv.	Conv.
Input:	$64 \times 64 \times 1$	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$
Parameters:	filter = 64 kernelsize = 4 stride = 2 pad = 1	filter = 128 kernelsize = 4 stride = 2 pad = 1	filter = 256 kernelsize = 4 stride = 2 pad = 1	filter = 512 kernelsize = 4 stride = 2 pad = 1	filter = 21 kernelsize = 4 stride = 1 pad = 0
Output:	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$	$1 \times 1 \times 21$
Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	-
Batchnorm.:	-	✓	✓	✓	-
Training Parameters		optimization algorithm: Adam; learning rate = 0.0003; batch size = 100;			

Table 5.4: Detailed description of the regression network trained on the NYU dataset to estimate the hand pose.

complexity for the following extensive analysis due to limited resources. Each of these annotations consists of three values, including the u,v position within the image and the depth. Therefore, the output layer consists of 21 neurons. The detailed network architecture is summarized in Table 5.4. Since layer 1 to layer 4 are identical to the discriminator and inference architecture, we could transfer the pretrained weights and use these layers either as fixed feature extractor, or for fine-tuning. Again, a fifth network is trained from scratch as reference. All networks are trained until convergence for different amounts of labeled training samples. After completion of the training process, we compute the mean Euclidean distance between the seven predicted joint locations and the ground truth annotations over the test set.

Figure 5.7 shows the results of this comparison. Similar to the head pose estimation experiment, initializing the network with the weights of the trained discriminator or inference net leads to significantly better results for small training sets. Again, the networks based on the inference net perform better than the discriminator-based. But, unlike the previous experiment, the randomly initialized network is not able to close the gap to the fine-tuned inference network. Also noticeable is that the randomly initialized network needs about $10,000$ labeled training samples to achieve the same performance as the model based on the inference net, trained with only 100 labeled training samples.

Similar to the head pose estimation, the second evaluation metric computes the fraction of correctly predicted test samples. Correct means that the distance between the predicted position and the ground truth is below a given maximal Euclidean distance for all joint locations. Figure 5.8 shows the results of this evaluation for a fixed training set of $10,000$ labeled samples. Initializing the regression network with the weights of the inference net, increases the fraction of correct samples from 15.56% to 45.5%, at a distance threshold of 50 mm.

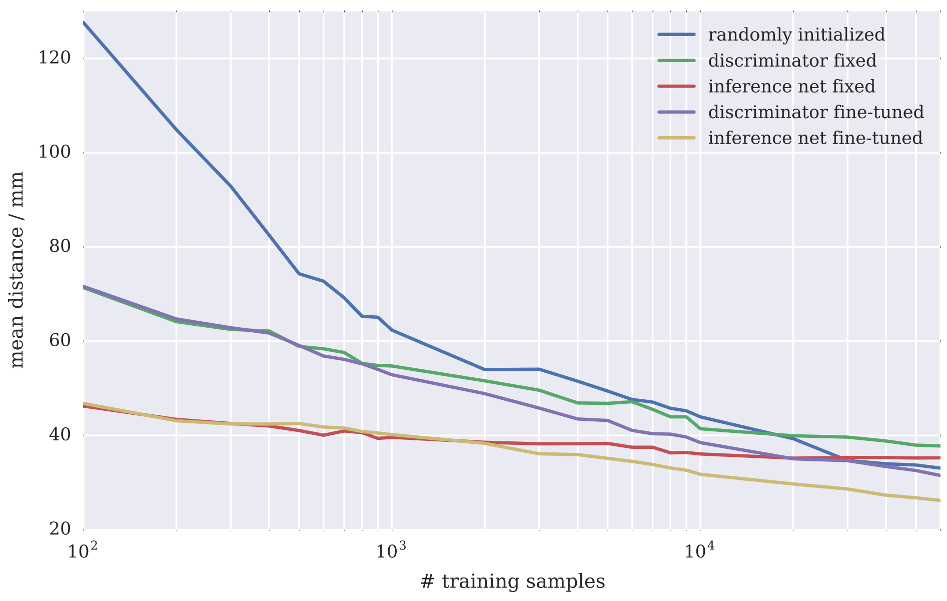


Figure 5.7: Mean Euclidean distance between predicted and ground truth joint locations on the NYU test-set as a function of the amount of labeled training samples for different unsupervised feature learning methods.

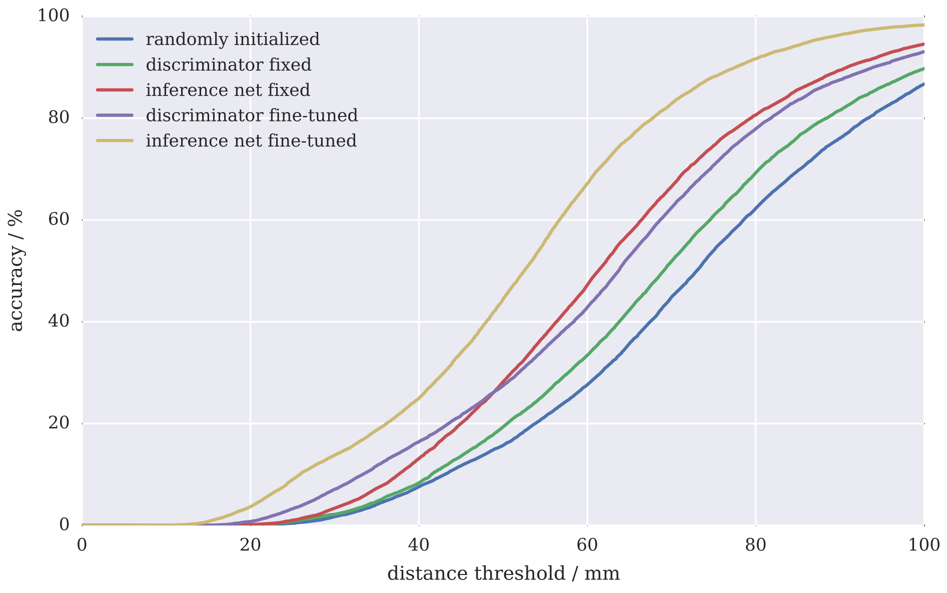


Figure 5.8: Accuracy of the hand pose estimation on the NYU dataset for a training set of 10,000 labeled samples and different unsupervised feature learning methods. The curves show the fraction of correct samples over an increasing success threshold.

5.2 Domain Specific Super Resolution

Aim of single image super resolution is to recover a high-resolution image from a single low-resolution input image. This classical computer vision problem is inherently ill-posed, since for every low-resolution pixel, a vast number of possible solutions exist. In practice, linear or bicubic interpolations are used to upsample an image. These methods are very fast but lead to blurry high-resolution images because they are not edge aware. More powerful methods are usually example based methods. These approaches learn a mapping from low- to high-resolution images and use it to create the most-likely high-resolution version of a given input image. These techniques are also known as image hallucination, since there is no guarantee that the high resolution details are equal to the (unknown) true details.

There are two classes of single image super resolution algorithms: generic and domain-specific methods. The former make no assumptions about the input and can be applied to all kinds of natural images, while the latter are designed for domain specific tasks, according to the used training dataset. In this Section we propose an upsampling approach based on *DCGANs* for face hallucination. Aim of this domain specific task is to upsample face images when the input resolution is very low. Solving this problem is especially difficult when the head poses and the illumination conditions vary. The idea is that the adversarial loss could increase the quality of the reconstructed images by incorporating characteristic details.

In the following we will first describe the dataset we use to train and test our network. Then, we present the network architectures and the training procedure and finally, we compare the results qualitatively and quantitatively.

5.2.1 CelebA Dataset

The CelebA dataset [34] is a large scale face dataset containing more then 200k celebrity images of over 10k identities. The color images cover large pose and illumination variations. We use the align & cropped version, where the images are aligned using a similarity transformation according to the eye locations and then resized to 218 × 178 pixel. The dataset is partitioned into two parts: a training-set containing 182,637 samples of nine thousand identities and a test-set containing the remaining 19,962 images of thousand identities. To augment the training-set to twice the size we mirror all samples horizontally.

During the preprocessing we crop out 115 × 115 patches covering only the face region and resize them to 64 × 64 pixel. The gray images are created by considering only the luminance channel in the YCrCb color-space. Finally, all samples are scaled into a range from -1 to 1. Figure 5.9 shows some resulting training samples.

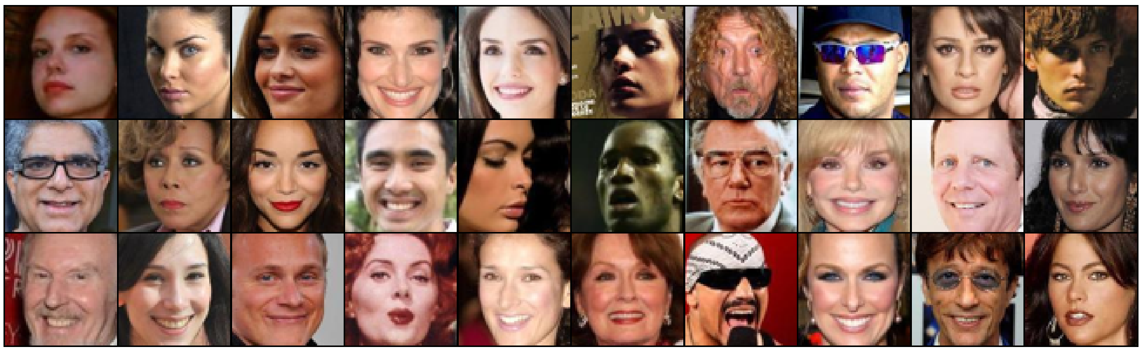


Figure 5.9: The CelebA dataset contains about 200k celebrity images covering different head poses and illuminations.

5.2.2 Network Architectures

We use three generator architectures, following different super resolution methods. In the first approach, deconvolution layers are used to upsample the low-resolution input image. In the second method, a deconvolutional network is used to predict a residual image that contains only the high-frequencies. Finally, in the last approach a convolutional network is used to synthesize this difference image from a coarse, mid-resolution input image.

5.2.2.1 Deconvolutional Network

The first method uses a deconvolutional network similar to the *DCGAN*, described in Section 3.4.3. The difference is, that the input to the generator is a low-resolution image instead of a noise vector. Similar to a conditional *GAN*, the generation process is therefore controlled by providing contextual information to the network. An example generator architecture is shown in Figure 5.10. Deconvolution layers are used to upsample the input image to the desired resolution. But in contrast to the *DCGAN*, intermediate convolution layers are used to increase the network depth. The kernel size and the padding parameter of these convolution layers have been chosen such that the spatial output dimensions are equal to the input dimensions.

The training procedure is shown in Figure 5.11. First, the original high-resolution image is down-sampled and used as input to the generator, which outputs the predicted high-resolution image. Like in a standard *GAN*, aim of the discriminator is to distinguish between the real image and the up-sampled image.

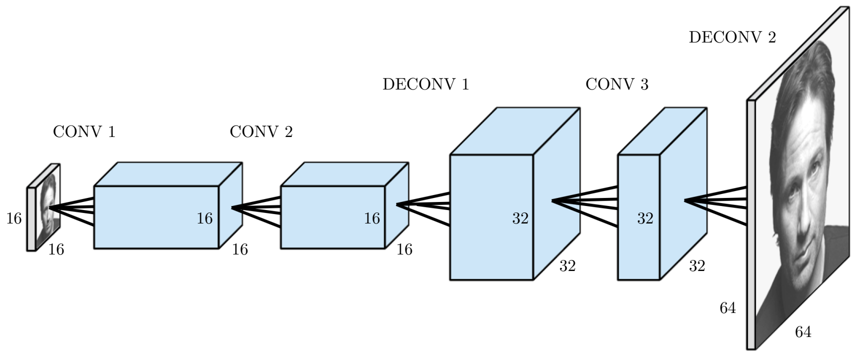


Figure 5.10: Super Resolution - Deconvolutional Network: The generator network uses deconvolution layers in combination with intermediate convolution layers to upsample the low-resolution input image.

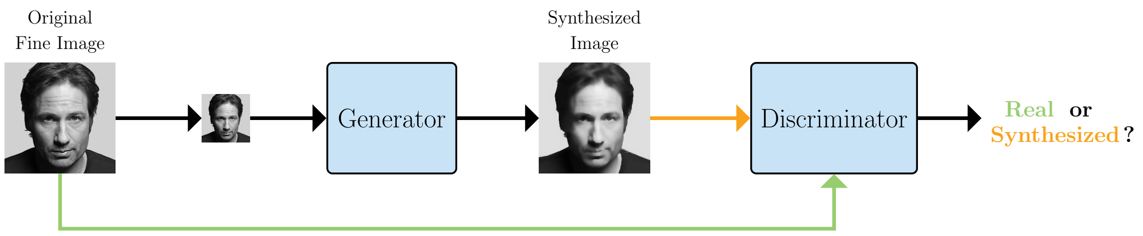


Figure 5.11: Super resolution training procedure: Input to the generator network is a downsampled version of the original image. Aim of the discriminator is to distinguish the upsampled image from the original high-resolution image.

5.2.2.2 Deconvolutional Network - Residual Image

The generator network in the first approach directly models the high-resolution image. Basically, a high resolution image can be decomposed into low- and high-frequency information. The first contains smooth variations and corresponds to the low-resolution image. The second part, also known as residual image, contains the image details. In the previously described approach, the generator has to reconstruct the image details, and in addition, carry the low-frequency information to the output. Another super resolution approach is to predict only the residual image containing the high-frequency part instead of predicting the final high-resolution image [48, 55]. Kim *et al.* [25] showed that this is also beneficial for neural network approaches. They argued that otherwise training time might be wasted to learn the low-resolution output and the convergence rate of learning the more important image details decreased.

Following this idea, we adapt the *GAN* framework, shown in Figure 5.12 as follows: Similar to the previous approach, the input to the generator is a downsampled version of the original high resolution image. But, this time the generator predicts only the image details. The network architecture is equal to the last generator (Figure 5.10). Aim of the

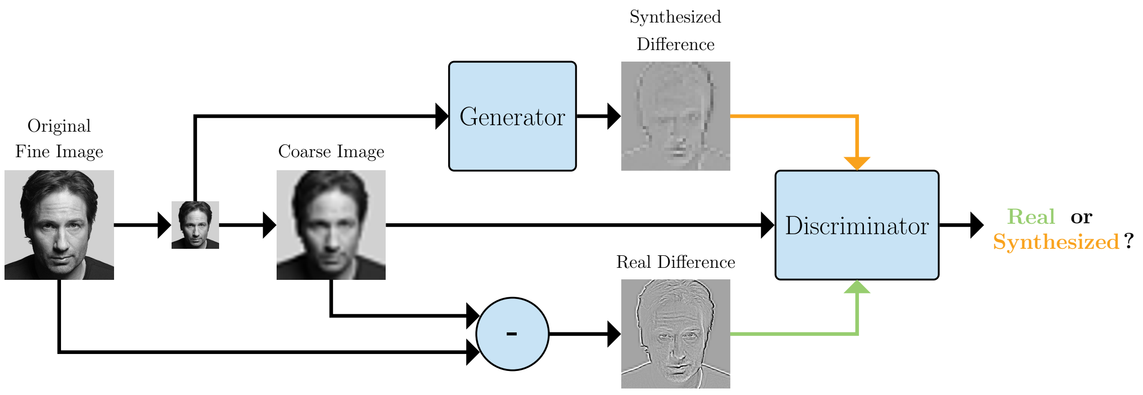


Figure 5.12: Super resolution training procedure (residual image): Input to the generator is a downsampled version of the original fine image. The discriminator, conditioned on the coarse image, tries to distinguish real image details from residuals synthesized by the generator.

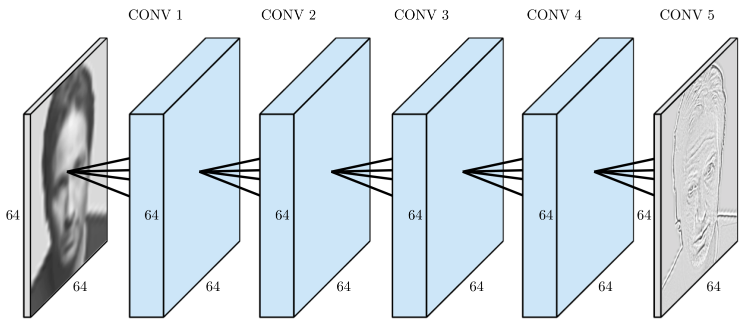


Figure 5.13: Super Resolution - Convolutional Network: Input to the generator is a coarse mid-resolution image of the same size as the desired output. Several convolutional layers of the same type are used to synthesize the corresponding residual image.

discriminator, which is conditioned on the coarse mid-resolution image, is to distinguish between real and synthesized residual images. The contextual information, the coarse image, is created by upsampling the low-resolution image using bicubic interpolation.

5.2.2.3 Convolutional Network - Residual Image

In this approach, we use a generator architecture similar to the network described by Kim *et al.* [25]. Input to this network is an interpolated coarse mid-resolution image with the same spatial dimensions as the desired output. This input is created by bicubic upsampling the low-resolution image. The network consists of several convolutional layers whereas all layers are of the same type (equal kernel-size & equal number of feature maps), as shown in Figure 5.13. Similar to the last method, the network is trained to output the residual image. Except the changed generator input, the training procedure is equal to the last approach.

5.2.3 Loss Function

The commonly used loss function to train a convolutional neural network for image super resolution is the Mean Squared Error (MSE) loss, given as

$$E_{MSE} = \frac{1}{N} \sum_{i=1}^{N} \| G(\mathbf{x}_i^{\mathrm{LR}}) \;-\; \mathbf{x}_i^{\mathrm{HR}} \|_2^2, \tag{5.2}$$

where $\mathbf{X} = \{(\mathbf{x}_1^{\mathrm{LR}}, \mathbf{x}_1^{\mathrm{HR}}), \ldots, (\mathbf{x}_N^{\mathrm{LR}}, \mathbf{x}_N^{\mathrm{HR}})\}$ is a training-set of N low- & high-resolution image pairs and $G(\cdot)$ is the upsampling network. The problem using this error function is, that it leads to blurry predictions since it does not preserve discontinuities like edges. If an output pixel has two equally probable values y_1 and y_2, then the *MSE* loss would lead to $y_{avg} = (y_1 + y_2)/2$, even if this output is very unlikely.

On the other hand, the loss to train a generator network within a *GAN* framework, see Section 3.2, is given as

$$E_{GAN} = -\frac{1}{N} \sum_{i=1}^{N} \log \left(D(G(\mathbf{x}_i^{\mathrm{LR}})) \right), \tag{5.3}$$

where $D(\cdot)$ is the discriminator network. In contrast to the *MSE* loss, this error function does not explicitly include the corresponding high-resolution image $\mathbf{x}_i^{\mathrm{HR}}$. The generators objective is to create a sample that the discriminator can not distinguish from real high-resolution face images, without taking the true high-resolution image into account. Since the generator is not restricted, this might lead to a strongly hallucinating upsampling network.

An obvious way to limit this behavior is to combine both loss functions [57]. The idea is that the *MSE* loss acts as regularization term, and the adversarial loss function increases the quality of the reconstructed images by incorporating characteristic details. The corresponding loss function is therefore given as

$$E = \; E_{MSE} \; + \; \lambda \; E_{GAN}. \tag{5.4}$$

In our experiments we will use all three error functions and compare the resulting images qualitatively to verify this approach.

5.2.4 Results

In the following we evaluate the performance of our approaches in combination with the earlier described loss functions for $\times 4$ and $\times 8$ upsampling on the celebA dataset. All

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Type	Conv.	Conv.	Conv.	Conv.	Conv.
Input:	$64 \times 64 \times 1(2)$	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$
Parameters:	filter = 64	filter = 128	filter = 256	filter = 512	filter = 1
	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4	kernelsize = 4
	stride = 2	stride = 2	stride = 2	stride = 2	stride = 1
	pad = 1	pad = 1	pad = 1	pad = 1	pad = 0
Output:	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$4 \times 4 \times 512$	$1 \times 1 \times 1$
Nonlinearity:	Leaky ReLU	Leaky ReLU	Leaky ReLU	Leaky ReLU	Sigmoid
Batchnorm.:	-	✓	✓	✓	-

Table 5.5: Detailed description of the discriminator network used in the super resolution *GAN*. For the residual generator networks, the discriminator is conditioned on the coarse mid-resolution image, incorporated as second input channel.

networks are trained using Adam as optimization algorithm with a learning rate of 0.0002 and a mini-batch size of 64. For all experiments we use a similar discriminator architecture, described in Table 5.5. They differ only in the number of input channels, since the conditional discriminators used to distinguish real residuals from synthesized image details, receive the coarse mid-resolution image as second input (see Figure 5.12).

5.2.4.1 Deconvolutional Network

In the first experiment, we analyze the performance of the deconvolutional network that directly predicts the high-resolution image, given the low-resolution input. The used network architectures for $\times 4$ and $\times 8$ upsampling are summarized in Table 5.6. Figure 5.14 shows the results for $\times 4$ upsampling. The two leftmost columns show the nearest-neighbor & bicubic uspampled input for comparison. In the next three columns, the results of the upsampling network in combination with the different error functions are shown. The ground truth high-resolution image is shown in the last column.

As expected, the bicubic interpolation results in very blurry images. The upsampling network, trained using the *MSE* loss, significantly increases the image quality. However, on closer examination it is apparent that the images are still slightly blurry, especially around the eyes or lips. On the other hand, the network trained using only the adversarial loss creates images with clear defined edges and enhances characteristic details. But, since this loss does not contain an explicit reconstruction term, the network tend to hallucinate more. Combining both error functions seems to represent a good compromise, where the images are sharp, but nonetheless close to the ground truth high-resolution image.

The results of the $\times 8$ upsampling network, shown in Figure 5.15 are similar to the first experiment. Again, incorporating the adversarial loss leads to cleaner edges and stronger accentuated face details. Due to the very low input resolution

(8 × 8 pixel), the amount of hallucination is increased. Especially the network optimized on the adversarial loss is strongly hallucinating, which sometimes result in unrecognizable identities. However, despite the very coarse input, the identities reconstructed by the *MSE* based networks are surprisingly close to the ground truth.

		Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**	**Layer 5**
4×	Type	Conv.	Conv.	Conv.	Conv.	Deconv.	Deconv.
	Input:	16 × 16 × 1	16 × 16 × 16	16 × 16 × 32	16 × 16 × 64	16 × 16 × 128	32 × 32 × 64
	Parameters:	filter = 16	filter = 32	filter = 64	filter = 128	filter = 64	filter = 1
		kernelsize = 3	kernelsize = 3	kernelsize = 3	kernelsize = 3	kernelsize = 4	kernelsize = 4
		stride = 1	stride = 1	stride = 1	stride = 1	stride = 2	stride = 2
		pad = 1	pad = 1	pad = 1	pad = 1	pad = 1	pad = 1
	Output:	16 × 16 × 16	16 × 16 × 32	16 × 16 × 64	16 × 16 × 128	32 × 32 × 64	64 × 64 × 1
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	-	-
8×	Type	Conv.	Conv.	Deconv.	Conv.	Deconv.	Deconv.
	Input:	8 × 8 × 1	8 × 8 × 128	8 × 8 × 256	16 × 16 × 128	16 × 16 × 128	32 × 32 × 64
	Parameters:	filter = 128	filter = 256	filter = 128	filter = 128	filter = 64	filter = 1
		kernelsize = 3	kernelsize = 3	kernelsize = 4	kernelsize = 3	kernelsize = 4	kernelsize = 4
		stride = 1	stride = 1	stride = 2	stride = 1	stride = 2	stride = 2
		pad = 1	pad = 1	pad = 1	pad = 1	pad = 1	pad = 1
	Output:	8 × 8 × 128	8 × 8 × 256	16 × 16 × 128	16 × 16 × 128	32 × 32 × 64	64 × 64 × 1
	Nonlinearity:	ReLU	ReLU	ReLU	ReLU	ReLU	tanh
	Batchnorm.:	✓	✓	✓	✓	✓	-

Table 5.6: Deconvolutional generator architecture for ×4 and ×8 uspampling.

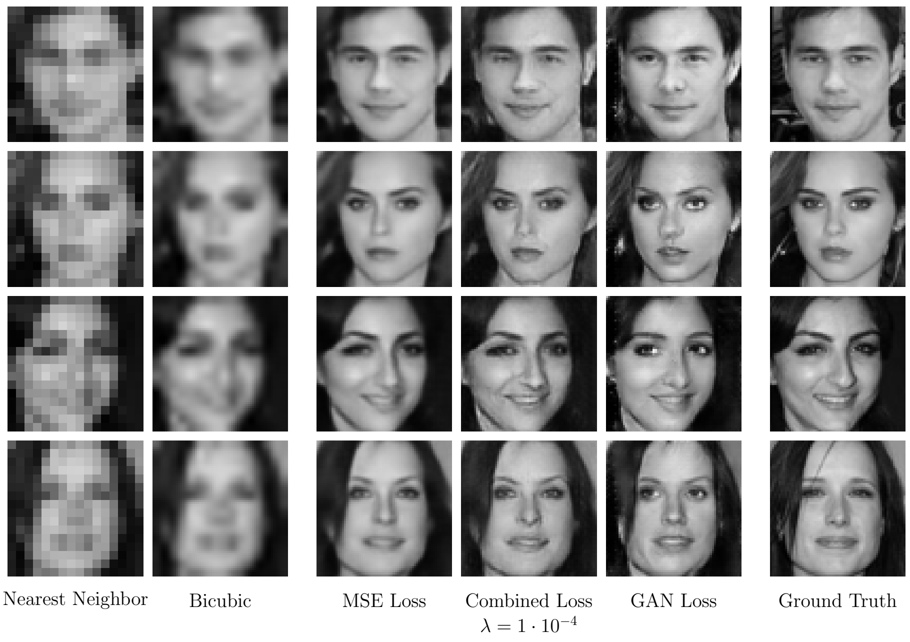


Figure 5.14: Results ×4 upsampling - Deconvolutional Network

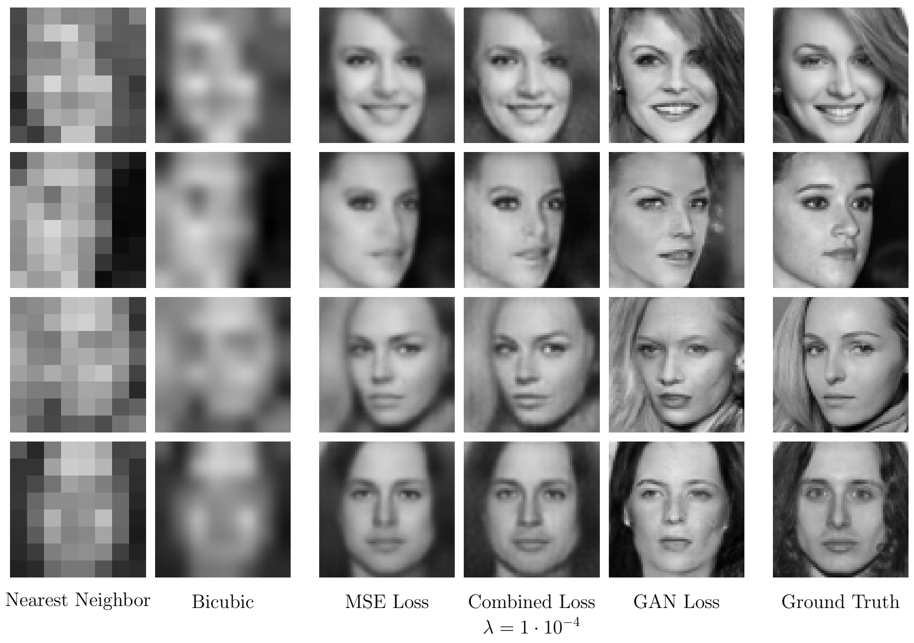


Figure 5.15: Results ×8 upsampling - Deconvolutional Network

5.2.4.2 Deconvolutional Network - Residual Image

In the next experiments we analyze the residual deconvolutional network. The generator architectures for ×4 and ×8 upsampling are similar to the last experiments and described in Table 5.6. As already mentioned, the discriminator is conditioned on the coarse mid-resolution image, see Table 5.5. The results for ×4 & ×8 upsampling, shown in Figure 5.16 and 5.17, are comparable with the non-residual network. However, the images upsampled by the adversarial networks seem to contain more high-frequency artifacts compared to the non-residual net. Since the network predicts only the difference to the high-resolution image, the amount of hallucination seems to be reduced.

5.2.4.3 Convolutional Network - Residual Image

The last network architecture we investigate uses convolutional layers to transform the bicubic upsampled coarse input image to the high-resolution output. The detailed generator architecture is summarized in Table 5.7. The results, shown in Figure 5.18 and 5.19, are slightly worse compared to the previous network architectures. While the upsampled images of the network trained on the *MSE* loss are comparable with the previous experiments, the adversarial networks produce significantly more high-frequency artifacts.

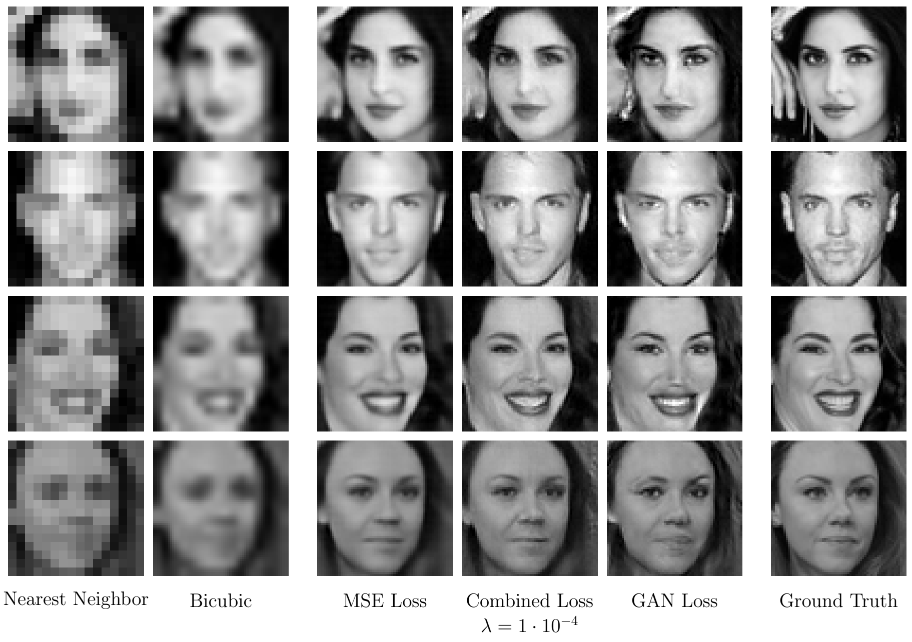


Figure 5.16: Results ×4 upsampling - Residual Deconvolutional Network

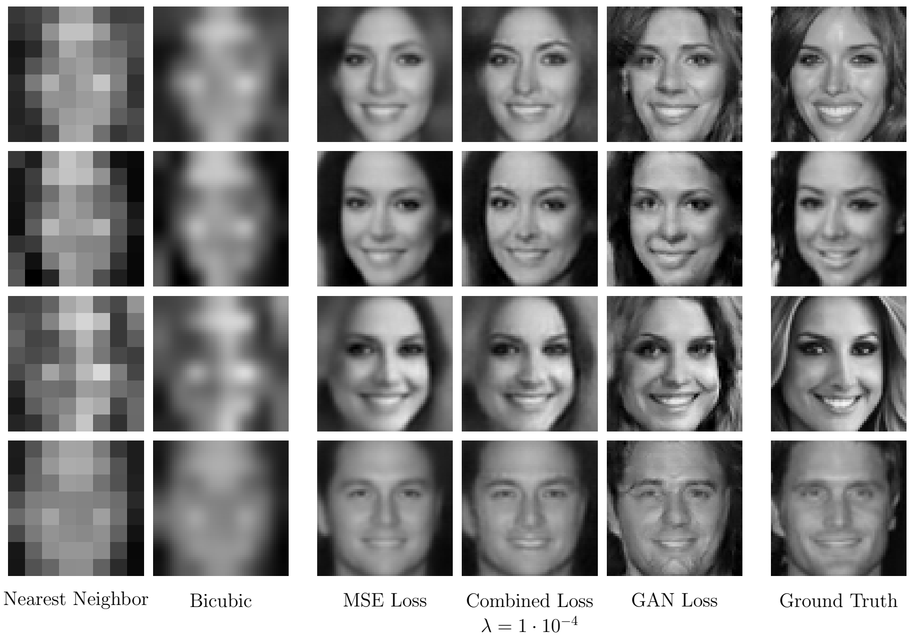


Figure 5.17: Results ×8 upsampling - Residual Deconvolutional Network

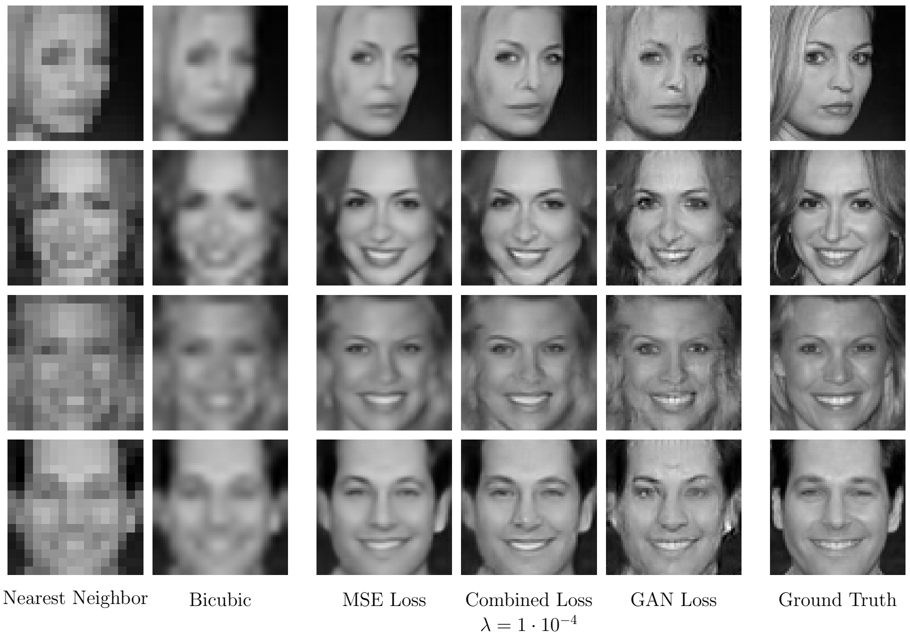


Figure 5.18: Results $\times 4$ upsampling - Residual Convolutional Network

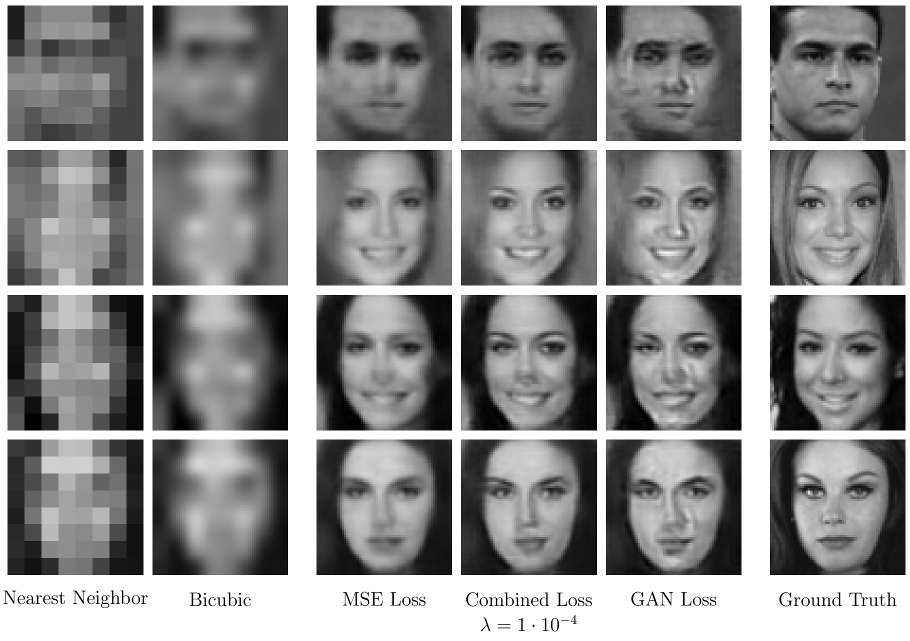


Figure 5.19: Results $\times 8$ upsampling - Residual Convolutional Network

	Layer 1	**Layer 2 - 8**	**Layer 9**
Type	Conv.	Conv.	Conv.
Input:	$64 \times 64 \times 1$	$64 \times 64 \times 32$	$64 \times 64 \times 32$
Parameters:	filter = 32	filter = 32	filter = 1
	kernelsize = 3	kernelsize = 3	kernelsize = 3
	stride = 1	stride = 1	stride = 1
	pad = 1	pad = 1	pad = 1
Output:	$64 \times 64 \times 32$	$64 \times 64 \times 32$	$64 \times 64 \times 1$
Nonlinearity:	ReLU	ReLU	-
Batchnorm.:	✓	✓	-

Table 5.7: Residual convolutional generator architecture for $\times 4$ and $\times 8$ uspampling

5.2.4.4 Qualitative Comparison

The previous experiments suggest that combining both error functions seems to represent a good compromise, where the images have clear defined face details but are nonetheless close to the ground truth high-resolution image. For a better comparison of the different architectures, Figure 5.20 shows upsampled images from all three approaches trained using the combined loss with $\lambda = 1 \cdot 10^{-4}$. The top rows show $\times 4$ upsampling results and the lower $\times 8$ usampled images. This comparison confirms the earlier findings. Differences in the upsampled images of the residual and the non-residual deconvolutional network are hard to find. Moreover, the high-resolution images of the convolutional residual network seem to be less sharp and contain more artifacts compared to the deconvolutional networks.

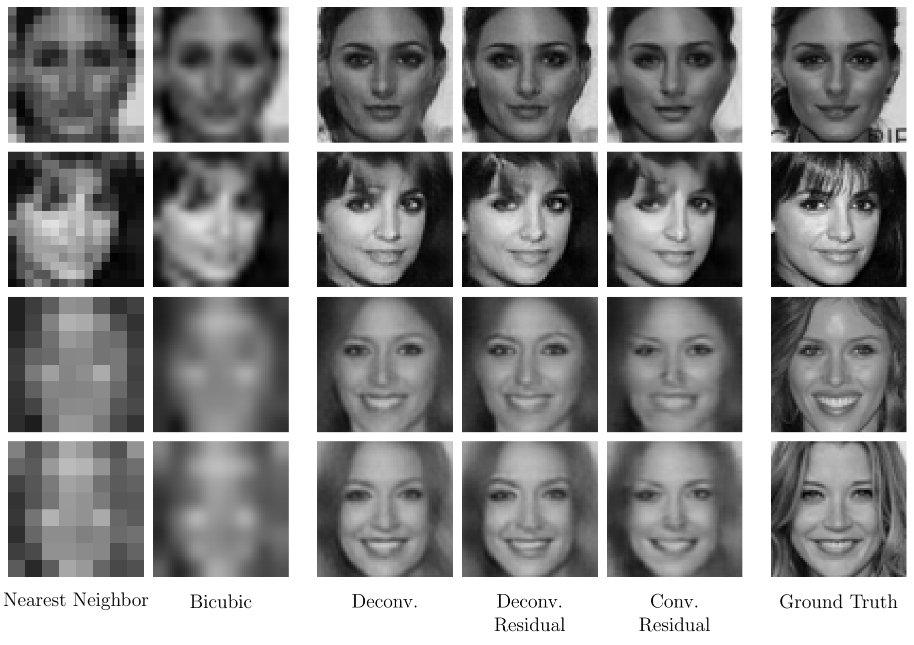


Figure 5.20: Architecture Comparison: All networks are trained using the combined loss with $\lambda = 1 \cdot 10^{-4}$. The top two rows show $\times 4$ upsampled images and the lower rows $\times 8$ upsampling results.

5.2.4.5 Quantitative Comparison

To quantitatively compare the different approaches, we compute the Peak Signal-to-Noise Ratio (PSNR) over the test-subset. This metric describes the ratio between the maximum of a signal and the corrupting noise and is usually used to measure the quality of a reconstructed image. The *PSNR* is given as

$$20 \cdot \log_{10} \left(\frac{MAX}{\|G(\mathbf{x}^{\mathrm{LR}}) - \mathbf{x}^{\mathrm{HR}}\|_2} \right), \tag{5.5}$$

where MAX is the maximum possible value, 255 for a 8-bit image. In addition we use a second metric called Weighted Peak Signal-to-Noise Ratio (WPSNR), which uses weights for perceptually different image areas and better matches the human perception [65].

The results of this comparison are summarized in Table 5.8 ($\times 4$ upsampling) and Table 5.9 ($\times 8$ upsampling). It can be seen that for both upsampling factors, the networks trained using the *MSE* loss achieve the highest *PSNR*, even though the combined loss leads to cleaner edges and sharper images. The first reason is that the *PSNR* is defined via the *MSE*. Therefore, this loss is the optimal error function for this criterion. However, this leads to blurry reconstructions in regions with strong ambiguity like edges (see Section 5.2.3). The second reason is that the adversarial loss only evaluates the image quality and does not contain a reconstruction term. Therefore, incorporating the adversarial loss increases the image quality, but leads to a stronger hallucinating network and reduces the *PSNR*.

As seen in the qualitatively results of the previous experiments, the networks trained using only the adversarial loss are strongly hallucinating, resulting in a significantly lower *PSNR*. The results for the *WPSNR* are very similar, since this metric also evaluates the difference to the ground-truth image. The slight changes of the identities due to the adversarial loss decreases the reconstruction score, although the images appear cleaner for the human observer.

	MSE Loss	Combined Loss	Adversarial Loss
	PSNR/WPSNR	PSNR/WPSNR	PSNR/WPSNR
Deconv.	26.52/29.36	25.47/28.36	21.19/18.88
Deconv. (Residual)	26.67/30.25	25.57/29.73	24.02/27.73
Conv. (Residual)	26.29/29.47	25.75/28.97	24.07/27.67

Table 5.8: Quantitative comparison of the different network architectures for $\times 4$ upsampling on the celebA test-subset. For comparison, bicubic upsampling leads to PSNR 23.92 / WPSNR 24.82

	MSE Loss PSNR/WPSNR	Combined Loss PSNR/WPSNR	Adversarial Loss PSNR/WPSNR
Deconv.	22.73/21.23	22.55/21.11	16.99/13.08
Deconv. (Residual)	22.87/21.49	22.56/21.19	19.86/18.46
Conv. (Residual)	21.84/19.87	21.72/19.84	21.24/19.39

Table 5.9: Quantitative comparison of the different network architectures for $\times 8$ upsampling on the celebA test-subset. For comparison, bicubic upsampling leads to PSNR 20.01 / WPSNR 16.83

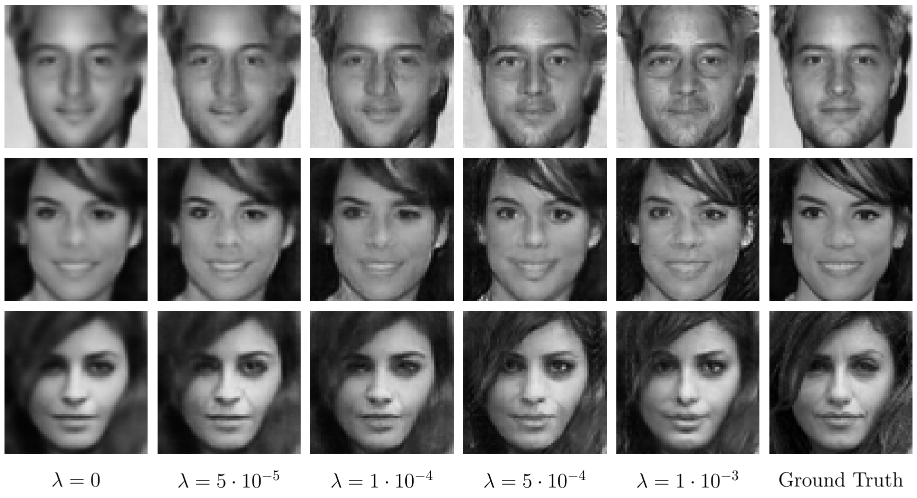

$\lambda = 0$
$\lambda = 5 \cdot 10^{-5}$
$\lambda = 1 \cdot 10^{-4}$
$\lambda = 5 \cdot 10^{-4}$
$\lambda = 1 \cdot 10^{-3}$
Ground Truth

Figure 5.21: To analyze the impact of the adversarial loss, we trained the non-residual deconvolutional network with different values for λ.

λ	0	$5 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$
PSNR [dB]	26.52	25.77	25.47	24.50	24.03

Table 5.10: Impact of the adversarial loss on the peak signal-to-noise ratio (PSNR).

5.2.4.6 Adversarial Loss Impact

To demonstrate the effects of incorporating the adversarial loss, we train the non-residual deconvolutional network with different values for λ. The results are shown in Figure 5.21. Similar to the previous experiments, the upsampled images of the network trained using only the *MSE* loss ($\lambda = 0$) are slightly blurry. As expected, increasing the weight of the adversarial loss improves the image quality. The images appear sharper and face characteristics are enhanced. This effect is particularly noticeable around the eyes and hair. But, increasing the weight of the adversarial loss leads also to a stronger hallucinating network and more high-frequency artifacts, hence the *PSNR* decreases as shown in Table 5.10.

5.2.4.7 Color Images

Since the human eye is less sensitive to chrominance than to luminance, a often used method to upsample color images is to process only the luminance channel Y in the YCbCr color space and bicubic upsample the chrominance channels. Another way is to use all three color channels as input to a network and directly upsample the color image. In order to compare these approaches, we train a non-residual deconvolutional network for color image upsampling. The architecture is similar to the network that processes only the luminance channel, see Table 5.6. The only difference is that this network has three input & output channels. We train the network to minimize the *MSE* error function and the combined loss with $\lambda = 1 \cdot 10^{-4}$. For comparison, we use the results from Section 5.2.4.1 and add bicubic-upsampled chrominance channels. The results for $\times 4$ and $\times 8$ upsampling are shown in Figure 5.22 and 5.23. At first glance, there is no difference between the two approaches. Similar to the previous experiments, incorporating the adversarial loss leads to sharper images and enhanced face details. But on closer examination, it is apparent that in a few samples, the bicubic interpolation of the chrominance channels modifies the eye or lip color, whereas the network is able to reconstruct these details.

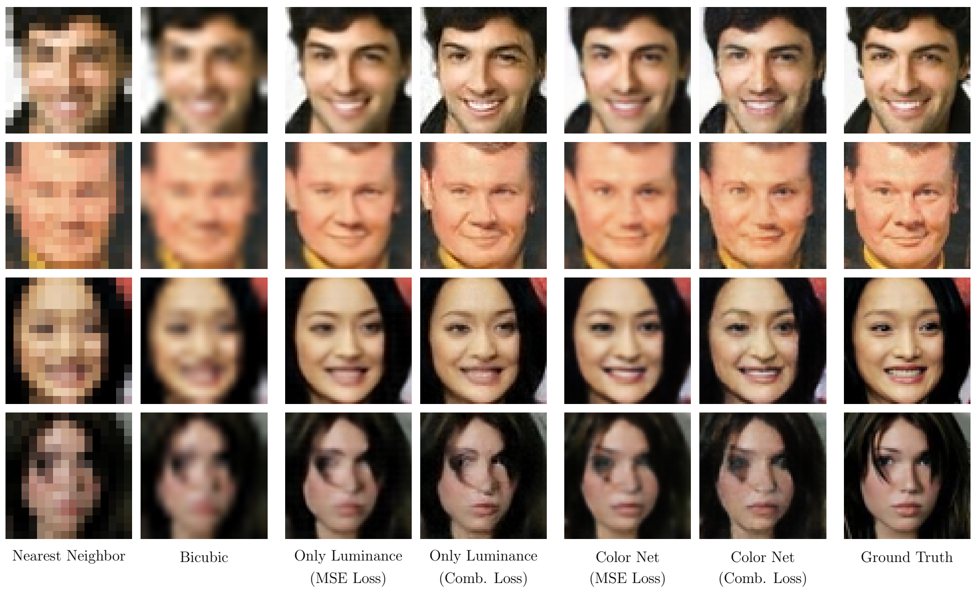


Figure 5.22: Results $\times 4$ color upsampling: The first method processes only the luminance channel and uses bicubic interpolation to upsample the chrominance channels. In the second approach the network directly predicts the color high-resolution image.

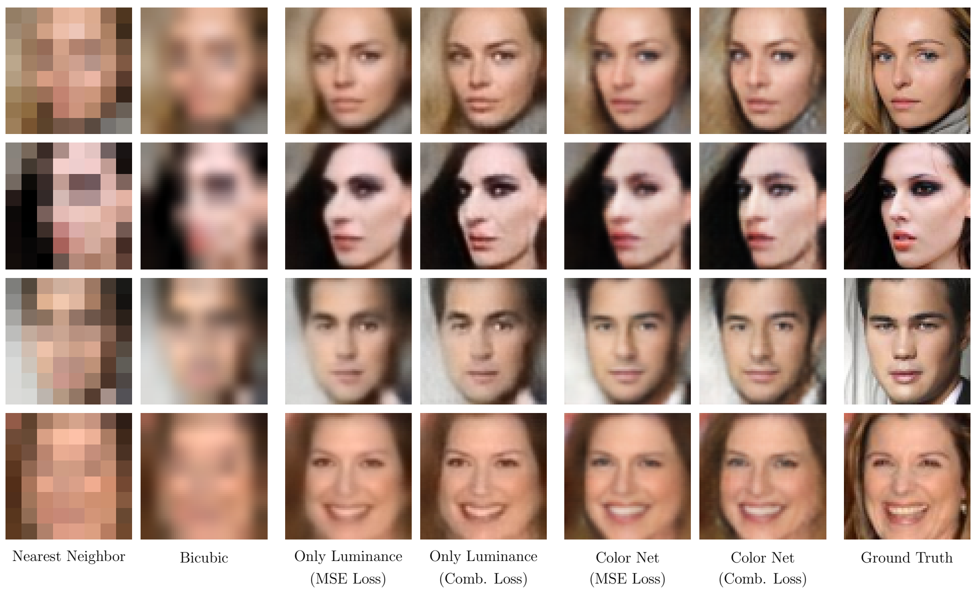


Figure 5.23: Results ×8 color upsampling

5.3 Summary

In this chapter the usability of Generative Adversarial Networks (*GANs*) to solve two fundamental computer vision problems was investigated. The first application was unsupervised feature learning where we reused parts of the trained discriminator network as initialization for the supervised task of interest. In addition, we used the data representations of a third network called inference net, which learned the inverse operation to the generator. The capabilities of these methods were evaluated for classification and regression tasks. In both cases performed the pre-initialized networks significantly better than a randomly initialized network when labeled samples were scarce.

In the second part *GANs* were applied to domain specific super resolution. We used non-residual & residual approaches, where either the high-resolution image was directly predicted or only the image details were synthesized. All networks were trained using three loss functions: the commonly used *MSE* loss, the adversarial loss and a combination of both functions. We showed that for both approaches, incorporating the adversarial loss leads to sharper images and increases the image quality by enhancing characteristic details. Summing up, these experiments demonstrated the high flexibility of adversarial nets and showed their potential for other computer vision applications.

6 Conclusion & Outlook

Generative Adversarial Networks (GANs) represent an attractive alternative to existing generative models based on maximum likelihood techniques, or related strategies. The advantage of adversarial nets is that the entire model is based on artificial neural networks. They can be trained using well studied techniques like backpropagation and no difficult sampling procedure is necessary. They perform well on various image datasets and are able to generate plausible natural images. However, the internal generation process of *GANs*, from the initial noise vector to the resulting image, as well as the learned data representations are relatively unexplored.

The aim of the first part of this thesis was to investigate the internals of adversarial nets more deeply and to identify ways to manipulate the generation process. We trained unconditional *GANs* on two depth-datasets and demonstrated that they were able to synthesize new realistic samples of human poses. Then, we explored the latent noise space and found that the relevant information for the generation process is rather encoded over multiple components than in individual values. One reliable way to identify semantic properties within this space was to average the noise vectors of synthesized samples sharing the same property. Moreover, by combining these directions using vector arithmetic it was possible to synthesize samples with desired properties, even though these networks were trained in an unsupervised manner. The second approach to control the generation process was to use conditional *GANs* where the ground-truth pose information was provided to both networks during the training. Using this method, the desired pose can be determined more precisely. However, the disadvantage of this method is that data annotations are necessary to train the *GANs*.

In the second part of this thesis, the applicability of *GANs* to solve two fundamental computer vision problems was investigated. First, we analyzed the potential of adversarial nets for unsupervised feature learning. Therefore, parts of the unsupervised trained discriminator network were reused as fixed feature extractor or as initialization for the supervised task of interest. In addition, a third network, called inference net, was trained to learn the inverse operation to the generator. The advantage is that this network can be trained on an infinite training set, since the generator can synthesize unlimited samples. Similar to the discriminator, the learned data representations of this network were reused for the subsequent supervised task. We evaluated the capabilities of these methods for classification and regression tasks. In both cases performed the pre-initialized networks significantly better than a randomly initialized network when labeled samples were scarce. Especially the good performances of the fixed feature extractors demonstrated that *GANs* are a strong candidate for unsupervised feature learning.

Finally, we applied *GANs* to face hallucination. The aim of this domain specific super resolution task is to upsample face images when the input resolution is very low. We used non-residual & residual approaches to upsample the input. The first method directly predicts the high-resolution image and the latter synthesizes only the image details, called residual image. All networks were trained using three loss functions: the commonly used Mean Squared Error (MSE) loss, the adversarial loss and a combination of both functions. We demonstrated that for both approaches, incorporating the adversarial loss increases the image quality compared to the *MSE* loss, since the images appear sharper and face characteristics are enhanced.

Summing up, we presented new insights into the generation process of *GANs*, proposed two ways to influence this process in order to synthesize depth-data with desired properties and demonstrated the universal applicability of this generative model. Future work could further explore operations in the latent noise space of unconditional *GANs* to control the generation process more precisely. This would avoid the need of large labeled datasets to train conditional *GANs*. The new data, synthesized in this manner, could then be used to augment the dataset for subsequent tasks. Moreover, there are several generative computer vision problems where adversarial nets might be beneficial. A possible application is image colorization, where the color version of a gray-scale image is hallucinated. Similar to the approach used for super resolution, both networks could be conditioned on the gray-scale input. The goal of the discriminator would be to distinguish between the real color channels and the color information synthesized by the generator. Another possible application for *GANs* is to predict future frames given a video sequence, where generator and discriminator could be conditioned on previous frames. This might also lead to data representations that could be useful for other supervised tasks, like action recognition in videos.

A List of Acronyms

CNN	Convolutional Neural Networks
DBM	Deep Boltzmann Machine
DBN	Deep Belief Network
DCGAN	Deep Convolutional Generative Adversarial Networks
GAN	Generative Adversarial Networks
i.i.d.	independent and identically distributed
ILSVRC	Imagenet Large Scale Visual Recognition Challenge
LAPGAN	Laplacian Generative Adversarial Networks
leaky ReLU	Leaky Rectified Linear Unit
MSE	Mean Squared Error
PSNR	Peak Signal-to-Noise Ratio
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit
SSD	Sum of Squared Differences
SVHN	Street View House Numbers
WPSNR	Weighted Peak Signal-to-Noise Ratio

Bibliography

[1] Alain, G. and Bengio, Y. (2014). What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research*, 15(1):3563–3593. (page 10)

[2] Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013). Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems*, pages 899–907. (page 10, 11)

[3] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*. (page 17)

[4] Denton, E. L., Chintala, S., and Fergus, R. (2015). Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in Neural Information Processing Systems*, pages 1486–1494. (page 41, 42)

[5] Doersch, C., Gupta, A., and Efros, A. A. (2015). Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1422–1430. (page 4)

[6] Dong, C., Loy, C., He, K., and Tang, X. (2015). Image Super-Resolution Using Deep Convolutional Networks. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PP(99):1. (page 5, 6)

[7] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159. (page 22)

[8] Efros, A. A. and Leung, T. K. (1999). Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038 vol.2. (page 5, 7)

[9] Fanelli, G., Dantone, M., Gall, J., Fossati, A., and Van Gool, L. (2013). Random Forests for Real Time 3D Face Analysis. *International Journal of Computer Vision*, 101(3):437–458. (page 47)

[10] Fischer, A. and Igel, C. (2012). An Introduction to Restricted Boltzmann Machines. *Lecture Notes in Computer Science: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, 7441:14–36. (page 8)

[11] Gauthier, J. (2014). Conditional generative adversarial nets for convolutional face generation. *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Winter semester*, 2014. (page 40)

[12] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587. (page 2)

[13] Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press. (page 2, 13, 24, 26)

[14] 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*, pages 2672–2680. (page 3, 33, 36, 37, 39, 40)

[15] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., and Wang, G. (2015). Recent Advances in Convolutional Neural Networks. *arXiv*, pages 1–14. (page 17)

[16] Hays, J. and Efros, A. a. (2008). Scene completion using millions of photographs. *Communications of the ACM*, 51(10):87. (page 6, 7)

[17] He, K., Zhang, X., Ren, S., and Sun, J. (2015a). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*. (page 29)

[18] He, K., Zhang, X., Ren, S., and Sun, J. (2015b). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034. (page 17)

[19] Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800. (page 9)

[20] Hinton, G. E., Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–54. (page 9)

[21] Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507. (page 8, 10)

[22] Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154. (page 23)

[23] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*. (page 23)

[24] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732. (page 2)

[25] Kim, J., Lee, J. K., and Lee, K. M. (2015). Accurate image super-resolution using very deep convolutional networks. *arXiv preprint arXiv:1511.04587*. (page 74, 75)

[26] Kingma, D. P. and Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13. (page 22, 50)

[27] Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. (page 37, 39, 42)

[28] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. (page 2, 17, 29, 30)

[29] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323. (page 37, 40)

[30] LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., and Huang, F. (2006). A tutorial on energy-based learning. *Predicting structured data*, 1:0. (page 8)

[31] LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer. (page 16)

[32] Lee, H., Ekanadham, C., and Ng, A. Y. (2008). Sparse deep belief net model for visual area V2. *Advances in Neural Information Processing Systems 20*, pages 873–880. (page 10)

[33] Lee, H., Grosse, R., Ranganath, R., Ng, A., and Honglak Lee, R. G. (2011). Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103. (page 4)

[34] Liu, Z., Luo, P., Wang, X., and Tang, X. (2015). Deep learning face attributes in the wild. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3730–3738. (page 72)

[35] Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440. (page 38)

[36] Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1. (page 17)

[37] 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 14)

[38] Mirza, M. and Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*. (page 40)

[39] 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-10)*, pages 807–814. (page 17)

[40] Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376. (page 21)

[41] 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*, volume 2011, page 4. Granada, Spain. (page 64)

[42] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*. (page 43, 45, 55)

[43] Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media. (page 13)

[44] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. (page 14)

[45] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1. (page 19, 21)

[46] Salakhutdinov, R. and Hinton, G. E. (2009). Deep boltzmann machines. In *International conference on artificial intelligence and statistics*, pages 448–455. (page 9, 10)

[47] Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229. (page 1)

[48] Schulter, S., Leistner, C., and Bischof, H. (2015). Fast and accurate image upscaling with super-resolution forests. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3791–3799. (page 5, 74)

[49] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. (page 27, 29)

[50] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Technical report, DTIC Document. (page 8)

[51] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*. (page 30, 43)

[52] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958. (page 22)

[53] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9. (page 29)

[54] Tieleman, T. and Hinton, G. (2012). Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*. (page 22)

[55] Timofte, R., Smet, V., and Gool, L. (2013). Anchored neighborhood regression for fast example-based super-resolution. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1920–1927. (page 74)

[56] Tompson, J., Stein, M., Lecun, Y., and Perlin, K. (2014). Real-time continuous pose recovery of human hands using convolutional networks. *ACM Transactions on Graphics (TOG)*, 33(5):169. (page 49)

[57] Tuzel, O., Taguchi, Y., and Hershey, J. R. (2016). Global-local face upsampling network. *arXiv preprint arXiv:1603.07235*. (page 76)

[58] Van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85. (page 30)

[59] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM. (page 10)

[60] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408. (page 10)

[61] Von Melchner, L., Pallas, S. L., and Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. *Nature*, 404(6780):871–876. (page 13)

[62] Vreeken, J. et al. (2002). Spiking neural networks, an introduction. *Institute for Information and Computing Sciences, Utrecht University Technical Report UU-CS-2003-008*. (page 14)

[63] Widrow, B., Hoff, M. E., et al. (1960). Adaptive switching circuits. (page 16)

[64] Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*. (page 17)

[65] Yang, C.-Y., Ma, C., and Yang, M.-H. (2014). Single-image super-resolution: a benchmark. In *Computer Vision–ECCV 2014*, pages 372–386. Springer. (page 83)

[66] Yu, F., Zhang, Y., Song, S., Seff, A., and Xiao, J. (2015). Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*. (page 44)

[67] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014*, pages 818–833. Springer. (page 30)