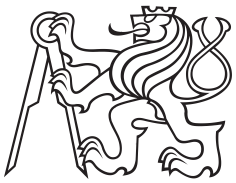
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Data Augmentation for Neural Networks Training

Antonín Vobecký

Supervisor: Mgr. Radoslav Škoviera, Ph.D.
Field of study: Open Informatics
Subfield: Computer Vision and Image Processing
May 2019

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Vobecký Antonín** Personal ID number: **434788**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer Vision and Image Processing**

II. Master's thesis details

Master's thesis title in English:

Data Augmentation for Neural Networks Training

Master's thesis title in Czech:

Rozšíření dat pro trénink neuronových sítí

Guidelines:

The objective of the diploma thesis is to propose, implement, test, and document a method for data augmentation. The target domain of the augmentation method is for image datasets dedicated to train artificial neural networks. For that reason, the evaluation of the proposed method should be done on a neural network. The intended application is in the area of autonomous driving. The augmentation should increase the variability of the original data and decrease the error rate of the neural network trained on the data.
The following steps should be accomplished:
1. Research existing data augmentation methods for neural networks.
2. Obtain a suitable dataset.
3. Propose and implement the individual components of a data augmentation system.
4. Choose a suitable neural network to be trained on the dataset and compare the error of the network with and without the augmented data.

Bibliography / sources:

[1] Wang, Ting-Chun, et al. "High-resolution image synthesis and semantic manipulation with conditional gans." arXiv preprint arXiv:1711.11585, 2017.
[2] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems, 2014.
[3] He, Kaiming, et al. "Mask r-cnn." 2017 IEEE International Conference on Computer Vision (ICCV), 2017.
[4] Johnson-Roberson, Matthew, et al. "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?." arXiv preprint arXiv:1610.01983, 2016.
[5] Duda et al. "Pattern classification." NY, USA 2001.

Name and workplace of master's thesis supervisor:

Mgr. Radoslav Škoviera, Ph.D., Robotic Perception, CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **09.01.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until: **30.09.2020**

Mgr. Radoslav Škoviera, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgements

I would like to thank my thesis advisor Mgr. Radoslav Škoviera, PhD. for an excelent guidance. I would also like to thank Ing. David Hurych, PhD. and Ing. Michal Uřičář, PhD. for their expertise consultations.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

V Praze, 23. May 2019

Abstract

The focus of this thesis is on techniques that can be used for the augmentation of an existing image dataset – specifically, a dataset containing humans in the individual images. For the purpose of the development, training, and evaluation of these techniques, a suitable dataset is obtained. First, current augmentation techniques based on image processing methods are explored, and their benefits are shown. Next, a novel framework for the conditional generation of images containing people is proposed. The ability to generate people in arbitrary, yet admissible, poses is beneficial for the training of any system involving human detection. It is especially important for the main motivation behind this work – the Autonomous Driving industry. Firstly, because the existing datasets are quite limited in the human pose and appearance variation. Secondly, because the strict safety requirements in autonomous driving applications call for the ability of validation in rare situations. In other words, it is advantageous to have the ability to sample from the huge distribution of admissible pose variants of pedestrians. The proposed approach allows generating images of people in a sensible required pose, specified via pose keypoints. It builds on top of the recent prevailing success of Generative Adversarial Networks [1]. The contributions comprise of a novel network architecture, as well as the novel loss terms specifically designed to generate visually appealing pedestrians seamlessly fitting the surrounding environment. The result of using such a network for augmentation of an existing dataset shows an increase of the resulting performance of the CNN based human detection system.

Keywords: machine learning, neural networks, training data

Supervisor: Mgr. Radoslav Škoviera, Ph.D.
CTU in Prague, CIIRC, Jugoslávských partizánů 1580/3, Praha 6

Abstrakt

Tato diplomová práce je zaměřena na techniky použité pro rozšíření stávajících trénovacích dat pro trénink neuronových sítí se zaměřením na datasety obsahující obrázky lidí. Pro účely vývoje, trénování, validace a testování byly vytvořeny vhodné datasety. Nejprve jsou zkoumány současné techniky na rozšiřování datasetu založené na metodách zpracování obrazu a přínosy těchto metod. Dále je navržen nový systém pro generování obrázků lidí. Schopnost generovat osoby v obecných, ale uvěřitelných pozach přináší výhody při učení systémů detekce lidí. Pro oblast autonomního řízení, která je hlavní motivací této práce, je toto ještě důležitější a to ze dvou hlavních důvodů. Zaprvé kvůli tomu, že stávající datasety jsou velice limitované co se póz lidí týče. Zadruhé je to pak existence přísných bezpečnostních předpisů pro aplikace autonomního řízení, které požadují validaci systémů v krajních situacích. Z toho vyplývá, že je velice výhodné mít systém, který umožňuje generovat obrázek člověka v zadané poze. Navržený systém je postaven na generativních soupeřících sítích (Generative Adversarial Networks). Mezi hlavní přínosy této práce patří inovativní architektura síti a nové ztrátové funkce navržené za účelem generování vizuálně přitažlivých obrázků osob. Výsledek použití takového systému pro rozšíření existujícího datasetu ukazuje navýšení výsledné úspěšnosti systému pro detekci chodců založeného na konvolučních neuronových sítích.

Klíčová slova: strojové učení, neuronové sítě, trénovací data

Překlad názvu: Rozšíření dat pro trénink neuronových sítí

Contents

Figures

Tables

Chapter 1

Introduction

Good data quality is the basis of every system based on statistical machine learning. These data should be a representative sample of the world, i.e., of all intended cases and scenarios. However, the data usually fail to meet these requirements. There are many reasons for such failure: limited time and resources for capturing and annotation, inability to identify all the use cases in advance, or just an inability to capture all the specific and rare situations that might arise.

The need for high-quality datasets is essential, especially in the autonomous driving industry, where there are high expectations to handle complex situations under all conditions, changing weather, time of the day, glare, etc. This is to ensure high safety and reliability of automated driving assistance systems (ADAS). Even nowadays, it is unfortunately not possible to capture all the required scenarios and conditions to train such a system. For autonomous driving and ADAS, it is very important to detect people in the car trajectory or in its close neighborhood. When capturing a suitable dataset for training such detector of pedestrians in driving scenarios, we will get very limited amount of real-world samples and almost zero situations where the pedestrian would be endangered by the car. One of the possible ways to solve this problem is to use data augmentation.

The goal of this thesis is to tackle the problems described above. First, the existing datasets and data augmentation techniques will be researched, and a suitable dataset for training and testing will be obtained. The next step will be a proposition and implementation of individual components of a data augmentation system. Finally, a suitable neural network to be trained on the dataset will be chosen, and it will be trained on the dataset with and without the use of the proposed augmentation techniques. The network trained on a dataset with the use of data augmentation should yield better results when compared to the network trained on a dataset without it.

Part I

Theoretical part

Chapter 2

Methodology

In this chapter, I will cover the underlying methodology used in this diploma thesis. First, I will talk about machine learning in general. This is well covered in books [36, 3, 37] and in [5]. Then I will follow with a more in-depth description of one of the branches of machine learning - artificial neural networks. Neural networks span a huge variety of models and; I will focus on Generative Adversarial Networks later in the chapter.

2.1 Machine Learning

Since the dawn of time, people try to make their life easier by using various tools and machines. Starting from firelock and the wheel, followed by plow and compass to more recently invented steam machines and airplanes, we, as mankind, arrived in the digital era with the invention of computers and the Internet. We can use computers to predict weather or stock prices, to recognize numbers or cat species or even to play computer games, compose music [38] and generate faces of nonexistent persons [30]. However, computers do not just *know* all of this. One of the ways to teach them these tasks is with the help of *machine learning*.

The term *Machine Learning* was first used by an American artificial intelligence researcher Andrew Samuel [39] back in 1959. A well-known definition is provided by Tom M. Michell [mitchell97mach]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

To show these three components in an example, let us imagine a system for reading words from a scanned document. The task T in this system would be to read the words correctly, performance measure P can be for example

an accuracy of read words compared to the ground truth and experience E would be pair of a scanned document and correct expected output of the system.

Another and more specific definition can be found in [5]:

> Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions.

Therefore, the author says that it presents the two central approaches to statistics: frequentist estimators and Bayesian inference.

In general, the machine learning algorithms can be divided into three major categories: supervised learning, unsupervised learning and reinforcement learning.

2.1.1 Supervised and semi-supervised learning

In supervised learning, the algorithm is provided with both inputs and desired outputs. In the case when some inputs do not come along with target outputs we talk about *semi-supervised learning*.

In the mathematical model of supervised learning, each training example is represented by a vector/matrix/tensor, and all the samples are represented by a *design matrix*. In the case when examples are represented as vectors, the design matrix has a shape of $N \times D$, where N is a number of samples and D is the dimensionality of examples. In other words, each row represents one example of the training dataset. Through an iterative optimization of an objective function (performance measure), supervised learning algorithms learn a function that can be used to predict the output associated with new inputs [37]. The goal of such learning is for the algorithm to be able to correctly determine the outputs for inputs that were not included in the training data. This ability is often called *generalization*.

One can divide the tasks solved with supervised learning algorithms to *classification* and *regression*.

Classification

In the task of classification, the goal of the algorithm is to assign each input vector to one of a finite number of discrete categories [3]. This task can be specified as learning a function $f\colon \mathbb{R}^n \to \{1, \ldots, k\}$ where n is the dimension of input vectors and k is number of categories.

A well-known classification task is the task of classifying images of numbers. A well known MNIST dataset [40] is usually used for as a training dataset for these tasks. An example of images from this set can be seen in Figure 2.1.

Figure 2.1: Example of size-normalized examples from MNIST dataset [2].

Regression

Regression differs from the classification in the fact that it learns to predict a numerical value. This can be described as a function $f: \mathbb{R}^n \to \mathbb{R}$ where n is the dimensionality of input samples.

Some examples of regression tasks are stock value prediction, house price prediction or temperature prediction.

2.1.2 Unsupervised learning

In unsupervised learning, the algorithm is provided with the inputs only and has no information about desired outputs. The goal of the unsupervised learning algorithms is mostly to find structure in the provided data. We can divide such algorithms to three basic categories: *clustering* and *density estimation*.

Clustering

The learning problem where the goal is to find groups of similar examples within the data is called clustering. Objects that belong to the same cluster are in some sense more similar to each other than to the objects in other clusters.

A well-known example of clustering is called *k-means clustering*. The goal of k-means clustering is to divide n samples to k clusters. Each cluster is

specified by the mean of samples belonging to that cluster. This mean is called *centroid*. All the samples are classified to the cluster with the nearest centroid. Such a partitioning of space corresponds to the partitioning into Voronoi cells. An example of k-means clustering into two classes ($k = 2$) is shown in Figure 2.2.

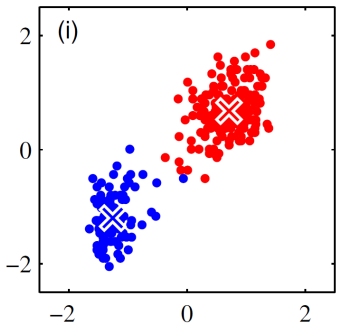


Figure 2.2: An example of result of the k-means clustering algorithm [3]

Density estimation

The task of density estimation can be seen as the construction of an estimate of an unobservable underlying probability distribution given observed data.

2.1.3 Semi-supervised learning

As one can imagine, in semi-supervised learning, the algorithm is provided with incomplete data. This means that a portion of inputs is missing the ground truth inputs.

2.1.4 Reinforcement learning

Reinforcement learning is concerned with the problem of finding suitable actions to take in a given situation in order to maximize a reward. A nice description of reinforcement learning can be found in [41]:

> Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error

search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning algorithms are well-known these days mostly because of their ability to play and solve various games and often even beat human players. The most famous are currently AlphaGo [42] and AlphaGoZero [43]

2.1.5 Tasks solved in machine learning

There are many other tasks solved in machine learning [5]. Here I will show a few more examples of such tasks.

Machine translation

The machine translation task aims to translate text or speech formulated in a source language into some other (target) language. The basic machine translation system would just translate each word in a sentence on its own without any context. This usually does not produce good results, e.g. because of the phrases contained in a language. Also, the word ordering in source and target language can be different. The training dataset in machine translation is often called *corpus*.

Statistical machine translation was one of the main approaches in the past. Just as the name says, it tries to generate translation using statistical methods based on bilingual text corpora.

In recent years, the area of machine translation research was overtaken by the rise of neural networks. Most of the translation systems (e.g. the one provided by Google) are nowadays implemented using neural networks.

Structured output

Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements [5].

An example of such a task is pixel-wise segmentation of images. In a pixel-wise segmentation, the algorithm is given an input image and is asked to assign one of categories to every pixel. This segmentation can be for example used to annotate the locations of roads in aerial photographs [44].

Neural networks overtook the area of image segmentation in recent years. One of the most suitable types of a neural network system for image segmentation are Fully Convolutional Networks (FCN) for Semantic Segmentation

[45]. You can find an example of resulting segmentation by this system in Figure 2.3.

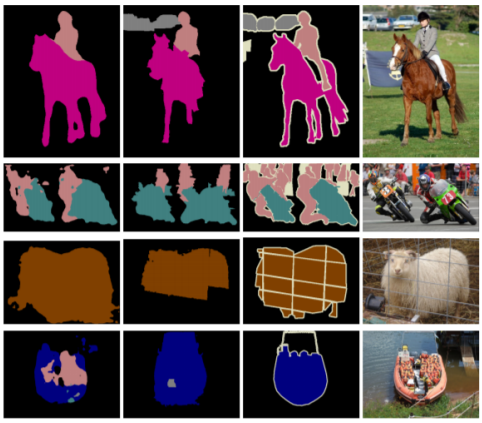


Figure 2.3: An example of segmentation by FCN in the first column and by an architecture called SDS [4] in the second column. In the third column is shown the ground truth segmentation and in the last one is an input image.

However, the tasks are not limited to have the same structure as the input. For example, in image captioning, the system is given an image as an input and is asked to describe its content by a natural sentence [46].

Synthesis and sampling

In the task of synthesis and sampling, the algorithm is asked to generate new samples that resemble those in the training data. Synthesis and sampling via machine learning can be useful for media applications where it can be expensive or tedious for an artist to generate large volumes of content by hand [5]. We can find such algorithms for example in computer games where they are used to generate for example textures for large landscapes.

Approaches that explicitly or implicitly model the distribution of inputs, as well as outputs, are known as ***generative models***, because by sampling from them it is possible to generate synthetic data points in the input space [3]. Given an observable variable X and a target variable Y, *generative model* is a model of the joint probability distribution on $X \times Y$, $P(X, Y)$.

One of the major ways in synthesis and sampling these days are Generative Adversarial Networks, GANs for shorts. GANs learn to mimic the distribution of training data via competitive game between two neural networks. One network aims to generate as realistic outputs as possible (usually called generator) and the goal of the other one is to tell the real samples from generated ones (this network is called the discriminator). I will describe GANs more deeply later on. GANs can be used for domain transfer [24] or super-resolution [47].

Denoising

In this type of task, the machine learning algorithm is given an input a corrupted example $\tilde{x} \in \mathbb{R}^n$ obtained by an unknown corruption process from a clean example $x \in \mathbb{R}^n$. The learner must predict the clean example x from its corrupted version $\tilde{x}$, or more generally predict the conditional probability distribution $p(x|\tilde{x})$ [5].

2.2 Artificial neural networks

Artificial neural network (ANN for short) is a widely used machine learning approach. Many of the current computer vision (CV) systems are based on ANNs. The research in CV is heavily focused on ANNs. The proposed solution in this thesis is not an exception.

Nowadays, this field is often connected with *deep learning*. Deep learning only appears to be new, because it was unpopular in the years that preceded its rise. In fact we can track deep learning back to 1940s when it was known as a subfield of *cybernetics* and in 1980s when it was called *connectionism*. It is known under the current name of deep learning since the year 2006.

The term *neural network* itself has origins in the attempts to find a mathematical model that would represent information processing in biological systems back in 1943 [48]. Despite the initial motivation behind constructing such models, artificial neural networks are not designed to be realistic models of biological function. They are most commonly used as models for statistical recognition.

The McCulloch-Pitts Neuron [48] was an early model of a biological neural network. The model was a linear one capable of recognizing two categories of inputs based on the test whether a function $f(\mathbf{x}, \mathbf{w})$, where $\mathbf{x}$ is an input and $\mathbf{w}$ are weights, is positive or negative. The big disadvantage of this model was that the weights had to be set manually by a human.

2.2.1 Perceptron

In the year 1958, Rosenblatt [49] described a model of *perceptron*. Perceptron is a two-class model in which the input $\mathbf{x}$ is first transformed using a fixed nonlinear transformation to give a feature vector $\phi(\mathbf{x})$, and then used to construct a generalized linear model of the form of equation 2.1

$$y(\mathbf{x}) = f\left(\mathbf{w}\top\phi\left(\mathbf{x}\right)\right) \tag{2.1}$$

where the nonlinear activation $f(.)$ is given by a step function of the form 2.2

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases} \tag{2.2}$$

The vector $\phi(\mathbf{x})$ typically includes a bias component $\phi_0(\mathbf{x}) = 1$ [3]. The output of $+1$ in the nonlinear activation described in equation 2.2 corresponds to the first class $\mathcal{C}_1$ and the output value of -1 corresponds to the other class $\mathcal{C}_2$. This implies the use of the $t \in \{\pm 1\}$ encoding of the target value.

In the training of perceptron, an error function called *perceptron criterion* is used. We seek a weight vector $\mathbf{w}$ such that patterns $\mathbf{x}_n$ in class $\mathcal{C}_1$ will have $\mathbf{w}^T\phi(\mathbf{x}_n) > 0$ and patterns in class $\mathcal{C}_2$ will have $\mathbf{w}^T\phi(\mathbf{x}_n) < 0$. With the use of the previously defined target encoding $t \in \{\pm 1\}$ we want all patterns to satisfy 2.3

$$\mathbf{w}\top\phi(\mathbf{x}_n)t_n > 0 \tag{2.3}$$

The perceptron criterion associates zero error with any pattern that is correctly classified, whereas for a misclassified pattern $\mathbf{x}_n$ it minimizes the quantity $\mathbf{w}^T\mathbf{x}_n t_n$. This results in perceptron criterion in the form 2.4

$$E_P(\mathbf{w}) = -\sum_{n\in\mathcal{M}} \mathbf{w}\top\phi(\mathbf{x}_n)t_n \tag{2.4}$$

where $\mathcal{M}$ is the set of misclassified patterns [3]. The contribution of the error associated with a misclassified pattern is a linear function of $\mathbf{w}$ in those regions of $\mathbf{w}$ where the pattern is misclassified - 2.3 is less than or equal to 0 - and is zero in regions where the pattern is classified correctly. This implies that the error function is piecewise linear [3].

To update the weights $\mathbf{w}$ of the perceptron, we apply the gradient descent (GD) to error function 2.4. In practice a stochastic gradient descent (SGD) is used. SGD and its modifications are commonly used for optimization in majority of neural networks. The update of weight vector $\mathbf{w}$ is given by equation 2.5

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{\tau} - \nu\Delta E_p(\mathbf{w}) = \mathbf{w}^{\tau} + \nu\phi_n t_n \tag{2.5}$$

where ν is the learning rate parameter that controls the speed of parameters change and τ is an integer corresponding to the step of algorithm. The learning rate parameter can be set equal to 1 since the perceptron function 2.1 is not affected by multiplication of weight vector $\mathbf{w}$ by a constant.

The perceptron algorithm follows these steps [50]

1. $\tau = 0$, $\mathbf{w}^{\tau} = 0$
2. Find a misclassified pattern $\mathbf{x}_i$. This corresponds to $\mathbf{w}^{\tau}\mathbf{x}_i \leq 0$.
3. If there are no misclassified petterns then terminate. Otherwise perform a weight update:
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{\tau} + \phi(\mathbf{x}_i) * t_i \tag{2.6}$$
4. Go to step 2.

The *perceptron convergence theorem* says that if there exists an exact solution (this means that the data set is linearly separable), then the perceptron

learning algorithm is guaranteed to find an exact solution in a finite number of steps.

Perceptron belongs to the family of linear models and thus inherits many limitations that these models have. The most famous problem is the fact, that these models are not able to learn the XOR function. This inability caused a backslash against biologically inspired learning in general [51] and was the first reason for the drop of the popularity of neural networks.

2.2.2 Artificial neuron

An artificial neuron is a name for a mathematical function that is used as a model of biological neurons. These neurons are building blocks of ANNs. Each neuron receives one or more inputs, weights them, sums them and then passes them through a non-linear function that is often called an *activation*. Different types of activation functions are shown in the later section 2.2.3.

The basic structure of an artificial neuron consists of a weight vector $\mathbf{w}$ and an activation function φ. The output of a neuron is then given by

$$y = \varphi\left(\mathbf{w}\top\mathbf{x}\right) \tag{2.7}$$

where usually $x_0 = 1$ and then the weight w_0 corresponds to *bias*. This structure is show in Figure 2.4

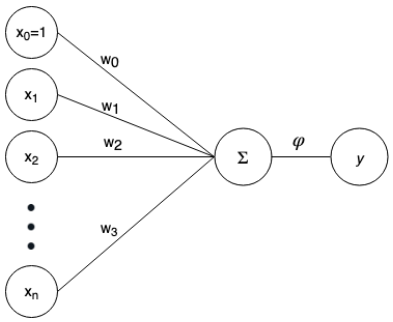


Figure 2.4: Basic structure of an artificial neuron.

2.2.3 Feed-forward neural networks

Feed-forward neural networks are the quintessential deep learning models. Their goal is to approximate some function f^* [5]. For a classifier, this means that function f maps input $\mathbf{x}$ to a category y. This can be written as $y = f^*(\mathbf{x})$. Feed-forward neural network aims to find such function $f(\mathbf{x}, \boldsymbol{\theta})$, where $\mathbf{x}$ is input and $\boldsymbol{\theta}$ represents parameters of the function, that would approximate the real function f^* the best.

The reason why are these models called feed-forward is in the fact that the information flows through the function being evaluated from an input

$\mathbf{x}$, through the intermediate computations, i.e., the hidden layers used to approximate the function f, and finally to the output y. There is *no feedback* in these networks.

One of the most important parts of every neural network are the activations. In section 2.2.3 I briefly introduce the most common ones.

Feed-forward neural networks include models such as single-layer perceptron (section 2.2.4), multi-layer perceptron (section 2.2.5) and convolutional neural networks (section 2.2.6). There are many more neural network models such as recurrent neural networks. However, such models are not used in this thesis and are not therefore described in the following sections.

Activation functions

An activation function is a function that is applied to a weighted sum of neuron inputs. The resulting value is then passed to the next node. There are many different activation functions and I will briefly describe few such functions that are most commonly used in neural networks. I will always show definition of the function, its derivative with respect to the input $\mathbf{x}$ and a plot of this function. The derivative of the activation function is important in the learning algorithm as I will show in the later section.

- *Sigmoid*
  Sigmoid (or logistic regression) is one of the first activations used in neural networks. It is a function described in equation 2.8, derivative w.r.t. $\mathbf{x}$ is in equation 2.9 and has a shape shown in Figure 2.5.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

$$f'(x) = f(x)\,(1 - f(x)) \tag{2.9}$$

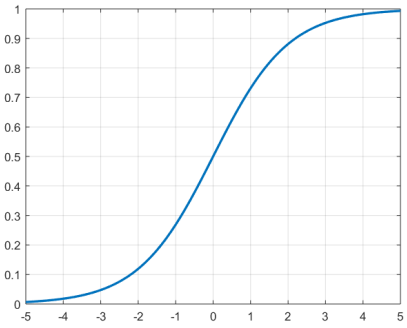


Figure 2.5: Sigmoid function.

- *Hyperbolic tangent*
  Hyperbolic tangent (tanh) is quite similar to the sigmoid function. Its range is $[-1, 1]$ instead of $[0, 1]$ for sigmoid. The advantage over the sigmoid is that he negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero with the tanh.

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.10}$$

$$f'(x) = 1 - f(x)^2 \tag{2.11}$$

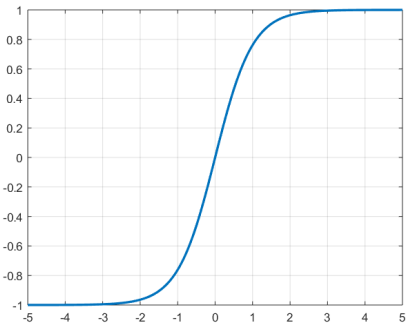


Figure 2.6: Hyperbolic tangent.

- *Rectifier linear unit (ReLU)*
  This activation function was first introduced in 2000 by Hahnloser [52] In the context of deep learning it was used for the first time in the year 2011 by Xavier Glorot, Antoine Bordes and Yoshua Bengio [53]. Deep sparse rectifier neural networks]. These days, it is the most commonly used activation function in the deep networks, mainly for its simplicity and because it does not saturate when activated.

$$f(x) = \max(0, x) \tag{2.12}$$

$$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{2.13}$$

- *Parametric ReLU*
  Parametric ReLU (PReLU) is a modification of ReLU. It differs in a way it deals with negative inputs. In the classic ReLU, the output for all negative inputs is 0. In the case of PReLU, the output for the negative input is the original input multiplied by a predefined coefficient a as it can be seen in the equation 2.14.

$$f(a, x) = \begin{cases} ax & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.14}$$

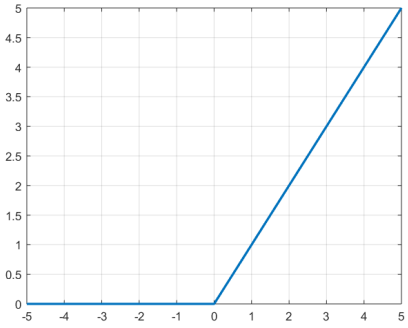


Figure 2.7: ReLU function.

$$f'(a,x) = \begin{cases} a & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{2.15}$$

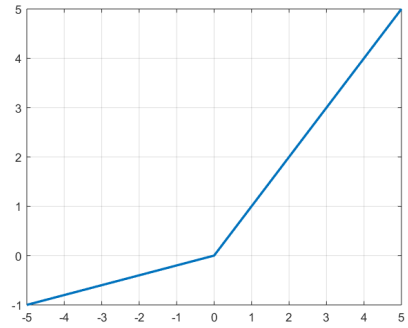


Figure 2.8: Parametric ReLU, $a = 0.2$.

- *Exponential linear unit (ELU)*
  ELUs try to make the mean activations closer to zero which speeds up learning. It has been shown that ELUs can obtain higher classification accuracy than ReLUs [54].

$$f(a,x) = \begin{cases} a(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.16}$$

$$f'(a,x) = \begin{cases} f(a,x) + a & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.17}$$

- *SoftPlus*
  SoftPlus is a smooth approximation to ReLU and is defined by equation 2.18. The comparison of these two functions can be seen in the Figure 2.11. The interesting thing about SoftPlus is that its derivative (eq. 2.19) is the sigmoid function (eq. 2.8).

$$f(x) = \ln(1 + e^x) \tag{2.18}$$

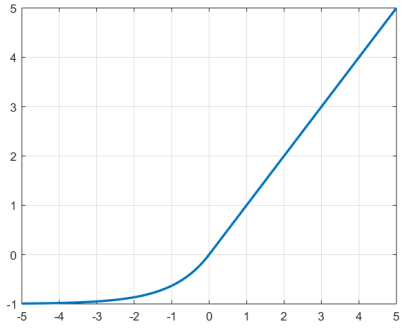


Figure 2.9: Exponential linear unit (ELU), $a = 1$.

$$f'(x) = \frac{1}{1 + e^{-x}} \tag{2.19}$$

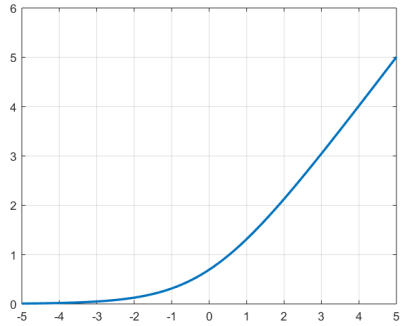


Figure 2.10: SoftPlus function.

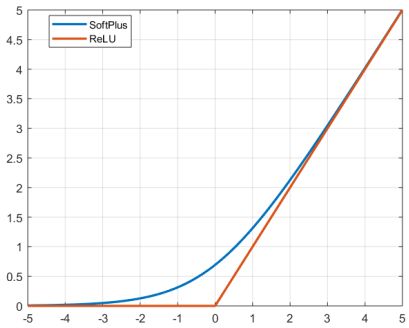


Figure 2.11: SoftPlus function vs ReLU.

- *Softmax*
  Softmax is used in the case of multinominal classification where do we have K mutually exclusive classes. It is defined as 2.20 with the

derivative written in 2.21

$$\hat{y}_k = \sigma(\mathbf{s}) = \frac{e^{s_k}}{\sum_{c=1}^{K} e^{s_c}} \tag{2.20}$$

$$\hat{y}'_k = \frac{\partial y_k}{\partial s_i} = \begin{cases} y_i(1 - y_i) & \text{for } i = j \\ -y_i y_k & \text{for } i \neq j \end{cases} \tag{2.21}$$

where $\mathbf{s} = (s_1, ..., s_K)$ is the input vector and $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_K)$ is the output vector.

Notice that the softmax function represents a probability distribution. This means that $\sigma_k \in (0, 1)$ for $k \in \{1, ..., K\}$ and $\sum_{c=1}^{K} \sigma_c(\mathbf{s}) = 1$.

2.2.4 Single-layer perceptron

The simplest possible neural network is a single-layer perceptron network (SLP for short). In this network, the inputs are fed directly to the outputs via a series of weights. The input is first multiplied by weights and then thresholded so that the neuron returns either $+1$ or -1. As it was written in section 2.2.1, such a network can learn linearly separable patterns.

A single-layer perceptron network can be used even for regression. The difference is in the activation function. One of the common activation functions is called *logistic function* or *sigmoid function*.

2.2.5 Multi-layer perceptron

A more complex network capable of distinguishing linearly inseparable data is called *multilayer perceptron* (or MLP for short). The biggest advantage of MLP over SLP is its ability to approximate more complex functions and can distinguish linearly non-separable data. As the name already says, this network consists of at least three layers of nodes

1. an *input layer*,
2. a *hidden layer*,
3. and an *output layer*.

2.2.6 Convolutional neural networks

All the previously described types of neural networks used fully-connected layers. The main difference in the convolutional neural networks (CNNs) is that it replaces those fully-connected layers by convolutional ones. CNN is a type of deep neural network that is usually used in computer vision and is used in this thesis as well. These networks have three basic mechanisms:

1. local receptive fields
2. weight sharing
3. subsampling

The structure incorporating the mechanisms written above is shown in Figure 2.12.

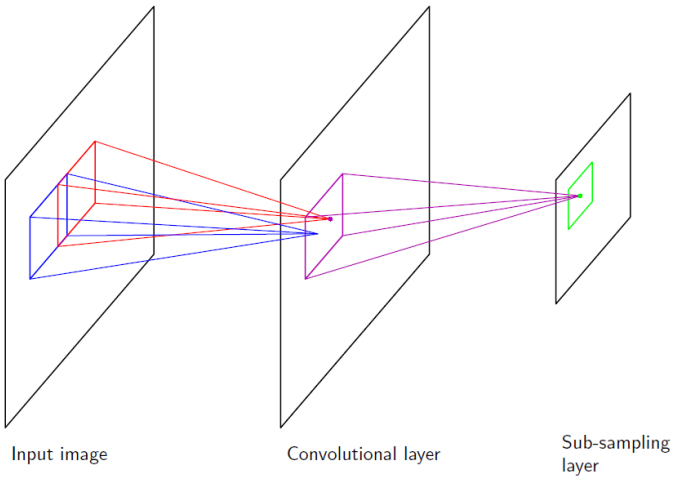


Figure 2.12: Diagram of a part of a CNN. You can see a convolutional layer followed by a subsampling layer [5]. The receptive fields are denoted as red and blue squares in the input image and as a purple one in the convolutional layer.

Convolution layer

As one might guess, the basic mechanism in convolutional neural networks is *convolution*. This convolution is implemented by *filter* applied to the input that results in a *feature map*. The fact that *the same* filter is used for the whole feature map results in *translation equivariance*. Usually there is more then one filter and therefore we get multiple feature maps as can be seen in Figure 2.13.

Each convolutional layer has two more hyperparameters (similar to standard mathematical convolution):

1. *Stride* - amount by which the filter shifts. An example is shown in Figure 2.14.
2. *Padding* - Zero padding pads the input volume with zeros around the border. This is often used in order to get the resulting output of the same spatial dimensions as the input. An example of zero padding is in Figure 2.15. For example, if we want to keep the spatial dimensions and

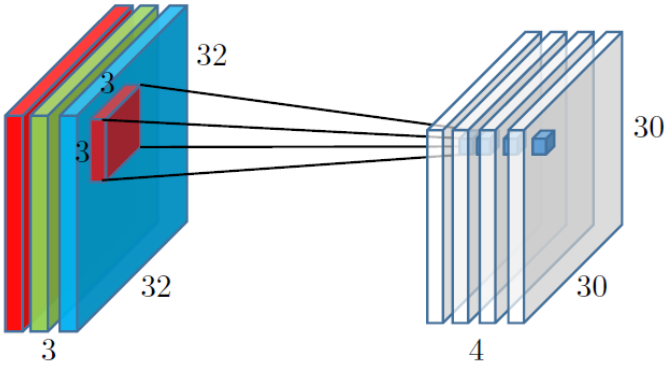


Figure 2.13: An example of four different filters of size 3×3 resulting in four feature maps [6].

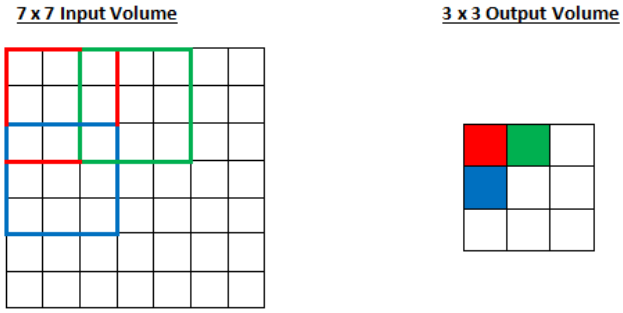


Figure 2.14: An example of 3×3 filter with stride 2 applied to 7×7 input resulting in the output of size 3×3 [7].

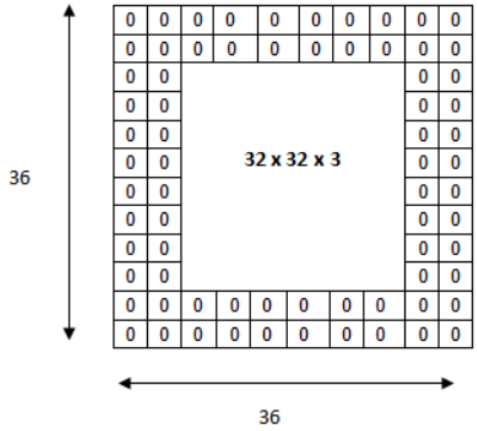


Figure 2.15: An example of zero padding with the value of 2 [7].

the convolution has a stride of 1, we set the zero padding to $P = \frac{F-1}{2}$, where F is the size of a side of the filter.

Knowing size of filter F, input width W_{IN}, stride S and padding P, we

can compute the output width W_{OUT} as shown in equation 2.22.

$$O = \frac{W - F + 2P}{S} + 1 \tag{2.22}$$

An example of a convolution operation performed on a 2D grid is shown in Figure 2.16. In this example, the input is of size $5 \times 5 \times 3$, there is a single 3×3 filter and a bias. This results in $3 \times 3^2 = 27$ parameters of the filter plus one parameter of bias. Each input channel is first convolved with respected filter weight matrix. Resulting intermediate feature maps are then summed up to form one feature map and the value of bias is added to each output. This results in the final feature map to which is then applied some *activation function*. An example of using ReLU activation to a feature map is show in Figure 2.17.

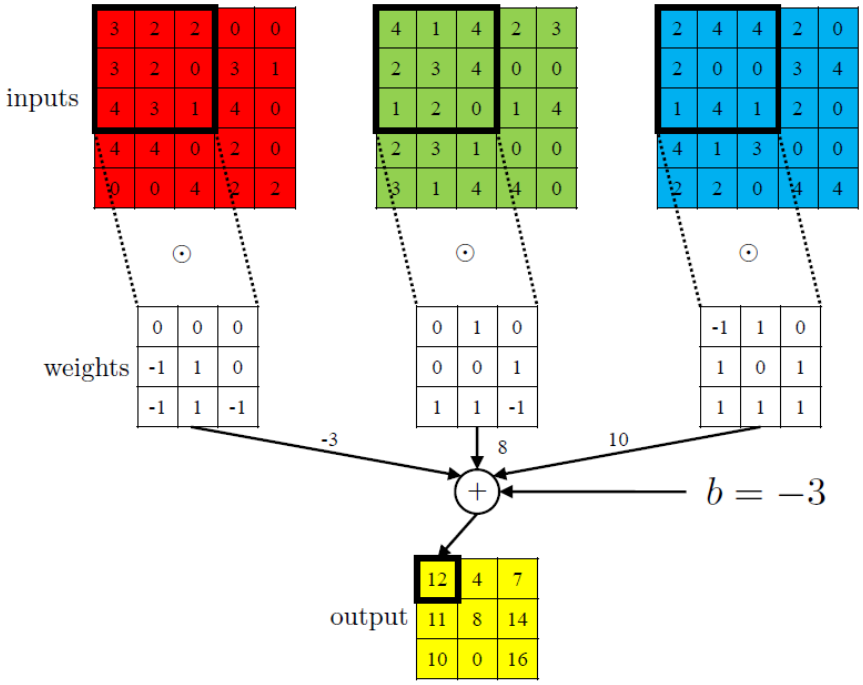


Figure 2.16: An example of a convolution operation on a 2D grid [6].

Pooling layer

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs [5]. It has hyperparameters *stride*, *padding*, and *filter size* that have the same meaning as in the case of a convolutional layer. There are multiple types of pooling functions.

1. *Max pooling* takes as an output a maximum over the input values. Example of this operation is in Figure 2.18

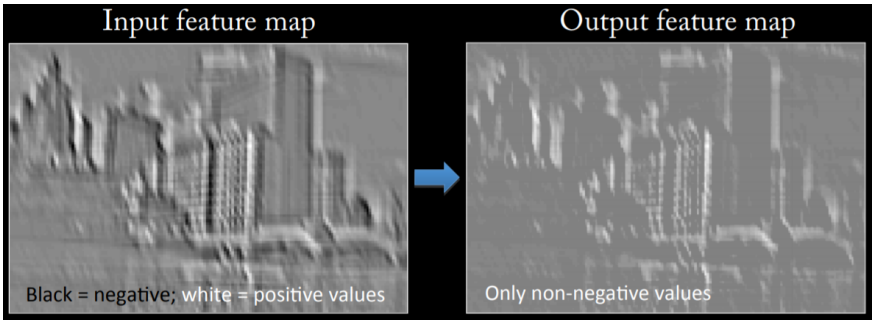


Figure 2.17: An example of ReLU activation used on a feature map [8].

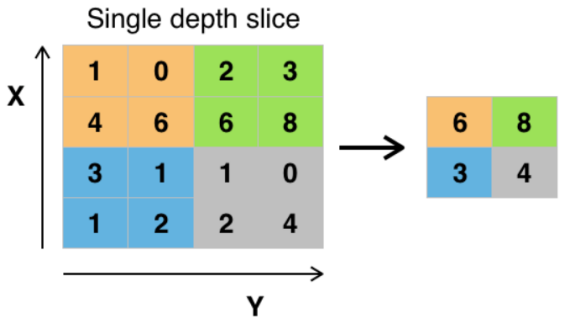


Figure 2.18: An example of max pooling with filter size of 2 and stride 2 [7].

2. *Average pooling* outputs the average of input values.

3. l_2*-norm pooling* computes the euclidean norm of inputs. This means that for inputs $x_1, ..., x_n$ it outputs $y = \sqrt{x_1^2 + ... + x_n^2}$.

The most common pooling filters are those of size 2 with stride 2. These filters downsample both width and height by a factor of 2. The pooling operations does *not* affect the depth of feature maps.

2.2.7 Learning in ANN

In this section, I will show you the basics of learning in ANNs. First, I will show common loss functions used for training, then introduce the idea of backpropagation, and finally describe optimization techniques used during the training.

Loss function

A loss function (or error function) is a function we want to minimize and gives us some notion about a quality of the output. This function compares the output $\hat{\mathbf{y}}$ of the neural network to the real target $\mathbf{y}$. There are many

different loss functions used in the training of neural networks. Here are the common ones:

- *Binary cross-entropy*
  The binary cross-entropy is often used in the case of binary classification. It is defined as in equation 2.23

$$L(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \ln\left(\hat{y}_i\right) + (1 - y_i) \ln\left(1 - \hat{y}_i\right) \right] \tag{2.23}$$

  where $\hat{\mathbf{y}}_i = f(\mathbf{x}_i, \mathbf{w})$ is the output of neural network represented as a function f with weights $\mathbf{w}$ and $\mathbf{y}_i$ is the corresponding ground truth value.

- *Multinominal cross-entropy*
  Multinominal cross-entropy is a modification of binary cross-entropy that can be used for multinominal classification. It is defined as 2.24

$$L(\mathbf{w}) = -\sum_{i=1}^{N} \sum_{c=1}^{K} y_{ic} \ln\left(\hat{y}_{ic}\right) \tag{2.24}$$

  where $\hat{\mathbf{y}}_i = (\hat{y}_{i1}, ..., \hat{y}_{ik})$ is the output of the neural network, $\hat{y}_{ic}$ corresponds to the output for c-th class where $c \in \{1, ..., K\}$ and $\mathbf{y}_i = (y_{i1}, ..., y_{ik})$ is the ground truth vector that is often represented as a one-hot vector.

- *Squared error ($\mathcal{L}_2$ loss)*
  Squared loss defined as 2.25 is often used in regression task and can be even used to compare images.

$$L(\mathbf{w}) = \sum_{i=1}^{m} \left(\hat{y}_i - y_i\right)^2 \tag{2.25}$$

  where $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_m)$ is the output of the neural network and $\mathbf{y} = (y_1, ..., y_k)$ is the vector of ground truth values.

- *$\mathcal{L}_1$ loss*
  $\mathcal{L}_1$ loss is quite similar to the squared loss and is defined as 2.26

$$L(\mathbf{w}) = \sum_{i=1}^{m} \left|\hat{y}_i - y_i\right| \tag{2.26}$$

  where once again $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_m)$ is the output of the neural network and $\mathbf{y} = (y_1, ..., y_k)$ is the vector of ground truth values.

Backpropagation

Backpropagation is a method used to compute a gradient of a loss function with respect to the parameters (weights $\mathbf{w}$) $\nabla L(\mathbf{w})$. This gradient is then used by optimization methods such as *gradient descent*.

The computation of a gradient $\nabla L(\mathbf{w})$ involves repetitive use of a chain rule. This computation can be further decomposed to the simplest possible modules [9].

Let $\delta^l = \frac{\partial L}{\partial \mathbf{z}^l}$ be the sensitivity of the loss to the module input for layer l, then

$$\delta_i^l = \frac{\partial L}{\partial z_i^l} = \sum_j \frac{\partial L}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l} \tag{2.27}$$

where $\mathbf{z}^l$ is the output of the l-th layer and $\mathbf{z}^l = (z_l^1, ..., z_l^n)$. From the equation 2.27 we see that we need to compute only the derivatives with respect to inputs [9].

In the case that the layer has some parameters, we want to know how does the loss change with respect to them. This can be computed as

$$\frac{\partial L}{\partial w_i^l} = \sum_j \frac{\partial L}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial w_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial w_i^l} \tag{2.28}$$

where $\mathbf{w}^l = (w_1^l, ..., w_m^l)$ are parameters of the l-th layer.

We can see an example of the diagram capturing these computations in Figure 2.19.

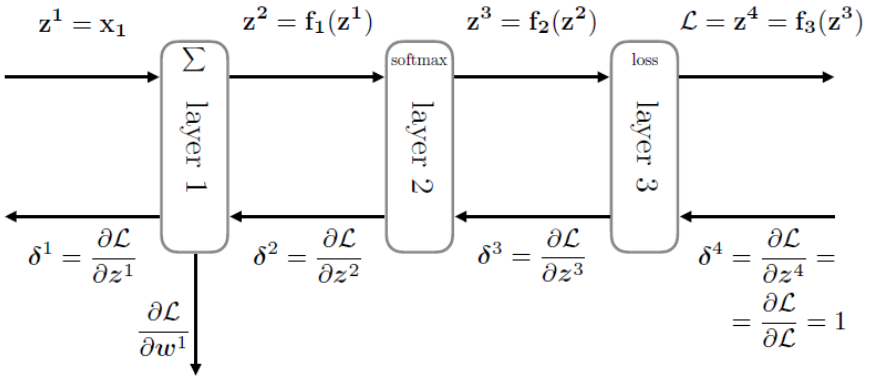


Figure 2.19: Diagram of backpropagation [9].

Optimization

In the optimization of neural network, the goal is to find such parameters $\boldsymbol{\theta}^*$ that would minimize the loss function. We can write this as 2.29

$$\boldsymbol{\theta}^* = \arg\min L(\boldsymbol{\theta}) \tag{2.29}$$

This optimization is most commonly performed via *gradient descent*. In general, gradient descent is a first-order iterative optimization algorithm that

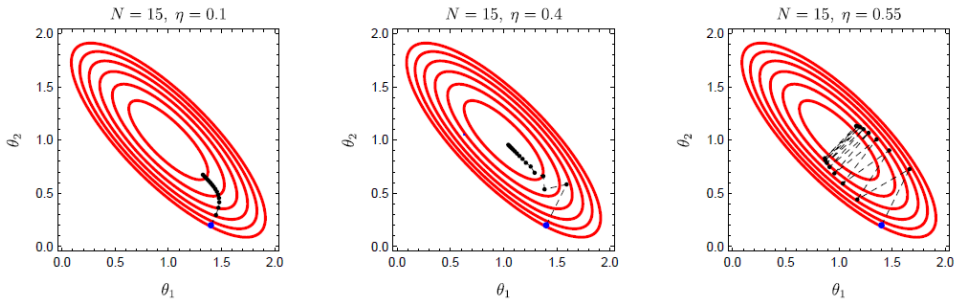


Figure 2.20: Comparison of gradient descent in two variables with different learning rates η [9].

aims to find the minimum of a given function. This algorithm is iteratively moving in the direction of steepest descent as defined by the negative of the gradient 2.30.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla L(\boldsymbol{\theta}^{(t)}) \tag{2.30}$$

where $\eta^{(t)} > 0$ is is a hyperparameter used to control the speed of change of the parameters in iteration t. It is often called *learning rate* or *step size*. In the Figure 2.20 you can see an example of gradient descent with $N = 15$ steps and 2 parameters θ_1 and θ_2 that should be optimized. The three plots that are shown differ in the learning rate η. In the first case $\eta = 0.1$ and we can see that the algorithm descents the space of the loss function nicely however quite slowly. On contrary, in the third case, the learning rate η is set to 0.55 which results in too drastic updates and the algorithm is not able to converge to the optima well. In the second case $\eta = 0.4$ and we can see a good trade-off between the update speed and precision.

There are multiple times when the weights can be updated [9].

- *(Full) Batch learning*
  In batch learning, the weights are updated once all the patterns are used (this defines an *epoch*). However, this approach is insufficient for redundant datasets.
- *Online learning*
  Online learning is a complete opposite to batch learning since in online learning the weights are updated after each training pattern. The noise can help overcome local minima but can also harm the convergence in the final stages while fine-tuning [9]. This update rule converges *almost surely* to local minimum when the learning rate $\eta^{(t)}$ decreases *appropriately* in time.
- *Mini-batch learning*
  In mini-batch learning, the weights are updated after a small sample of patterns.

One of a commonly used improvements to gradient descent is called *momentum*. It simulates inertia that helps to overcome possible plateaus in the

error landscape. The update rule is then performed in two steps

$$\begin{aligned} \boldsymbol{v}^{(t+1)} &= \mu \boldsymbol{v}^{(t)} - \eta^{(t)} \nabla L \left(\boldsymbol{\theta}^{(t)} \right) \\ \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} + \boldsymbol{v}^{(t+1)} \end{aligned} \tag{2.31}$$

where $\mu \in [0, 1]$ is the *momentum parameter*.

Adagrad. Adagrad is a shortcut of *Adaptive Gradient Algoritm*. This algorithm was first proposed in [55]. The motivation behind is that a magnitude of gradients differs a lot for different parameters. Thus this algorithm adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data [56]. The update rule can be expressed as

$$\begin{aligned} g_i^{(t+1)} &= g_i^{(t)} + \left(\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}} \right)^2 \\ \theta_i^{(t+1)} &= \theta_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t+1)}} + \epsilon} \cdot \frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}} \end{aligned} \tag{2.32}$$

where g_i accumulates squared partial derivatives w.r.t. the parameter θ_i, ϵ is a small positive number to prevent division by zero. The main disadvantage of Adagrad is that the ever-increasing g_i can lead to infinitesimally small learning rate and thus to slow convergence.

RMSProp. *Root Mean Squared Propagation* (RMSProp) is similar to Adagrad with the implementation of a moving average. It changes the way of computing g_i to

$$g_i^{(t+1)} = \gamma g_i^{(t)} + (1 - \gamma) \left(\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}} \right)^2 \tag{2.33}$$

which doesn't get the updates infinitesimally small as in the case of Adagrad.

Adam. *Adaptive Moment Estimation* (Adam) [57] is currently the most commonly used optimization algorithm in the training of neural networks. Adam combines the advantages of Adagrad and RMSProp.

Instead of adapting the parameter learning rates based on the average first moment as in RMSProp, Adam also makes use of the average of the second moments of the gradients. Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters β_1 and β_2 control the decay rates of these moving averages [58]. The algorithm is described in the original paper [57] as

1. $g_t = \nabla f_t(\theta_{t-1})$, get gradients
2. $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$, update biased first moment estimate

3. $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$, update biased second raw moment estimate
4. $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$, compute bias-corrected first moment estimate
5. $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$, compute bias-corrected second raw moment estimate
6. $\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}}+\epsilon}$, update parameters

The authors of [57] say the following about the algorithm: "The algorithm updates exponential moving averages of the gradient (m_t) and the squared gradient (v_t) where the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages. The moving averages themselves are estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance) of the gradient. However, these moving averages are initialized as (vectors of) 0's, leading to moment estimates that are biased towards zero, especially during the initial timesteps, and especially when the decay rates are small (i.e. the βs are close to 1). The good news is that this initialization bias can be easily counteracted, resulting in bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$."

Authors also claim that "empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods" and conclude that "using large models and datasets, we demonstrate Adam can efficiently solve practical deep learning problems."

2.2.8 Feature Normalization

Feature normalization is an important part of the training of artificial neural networks and is often crucial for effective training. In this section, I will briefly describe a few commonly used techniques of feature normalization.

Batch Normalization

Batch normalization [59] is a method of adaptive reparametrization, significantly reduces the problem of coordinating updates across many layers and can be applied to any input or hidden layer in the network.

It was initially proposed to solve internal covariate shift [59]. Recently, the authors of [60] showed that the batch normalization does not reduce internal covariate shift, but rather smooth the objective function which leads to the improved performance. The batch normalization also works as a regularizer in a network, helps it to generalize better and therefore it is unnecessary to use dropout.

Let us have a minibatch $\mathcal{B} = \{x_1, ..., x_n\}$ of training samples. The batch normalization works as follows:

- Compute minimatch mean $\mu_{\mathcal{B}}$

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.34}$$

- Compute minibatch variance $\sigma^2_{\mathcal{B}}$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{2.35}$$

- Normalize samples

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{2.36}$$

  where ϵ is a small constant added for numerical stability.

- Scale and shift

$$y_i = \gamma \hat{x}_i + \beta \tag{2.37}$$

  where γ and β are parameters that are learnt during the optimization process.

Layer Normalization

Layer normalization [61] addresses the problem of batch normalization that lies in need for large mini-batches for an accurate statistics estimation.

In layer normalization, the features are normalized within each sample instead across mini-batch (as in batch normalization). Like in the batch normalization, each neuron is given its own adaptive bias and scale.

The layer normalization statistics over all hidden units in the same layer are computed as:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} x_i^l \tag{2.38}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H} \left(x_i^l - \mu^l\right)^2} \tag{2.39}$$

where x_i^l denotes the input to the i-th hidden unit in the l-th layer and H denotes the number of hidden units in a layer. The normalized inputs are then computed as:

$$\hat{x}_i^l = \frac{x_i^l - \mu^l}{\sqrt{\sigma^2 + \epsilon}} \tag{2.40}$$

where ϵ is a small constant used for the numerical stability.

Instance Normalization

Instance normalization [62] is similar to the layer normalization. It normalizes the features within each channel in each training example.

For an input tensor of a shape $x \in \mathbb{R}^{T \times C \times W \times H}$, where x_{tijk} denotes its $tijk$-th element (k and j are spatial dimensions, i is the feature channel and t is the index of the image in the mini-batch), we first compute the statistics for each instance and channel as:

$$\mu_{ti} = \frac{1}{HW} \sum_{l=1}^{W} \sum_{m=1}^{H} x_{tilm} \tag{2.41}$$

$$\sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^{W} \sum_{m=1}^{H} (x_{tilm} - mu_{ti})^2 \tag{2.42}$$

The normalized inputs are then computed as:

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}} \tag{2.43}$$

Group Normalization

Another feature normalization technique is called group normalization (GN) [10]. GN divides the channels into groups and computes within each group the mean and variance for normalization.

GN performs the normalization in the standard way:

$$\hat{x}_i = \frac{1}{\sigma_i} (x_i - \mu_i) \tag{2.44}$$

where the index $i = (i_N, i_C, i_H, i_W)$ with N denoting the order in the batch, C in the channel and H and W are the spatial height and width. Parameters μ_i and σ_i are computed as:

$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k \tag{2.45}$$

$$\sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon} \tag{2.46}$$

where $\mathcal{S}_\rangle$ is the set of size m of pixels over which the mean and standard deviation are computed. The set S_i is determined as:

$$\mathcal{S}_i = \left\{ k | k_N = i_n, \left\lfloor \frac{k_C}{C/G} \right\rfloor = \left\lfloor \frac{i_c}{C/G} \right\rfloor \right\} \tag{2.47}$$

The comparison of these feature normalization techniques can be seen in Figure 2.21.

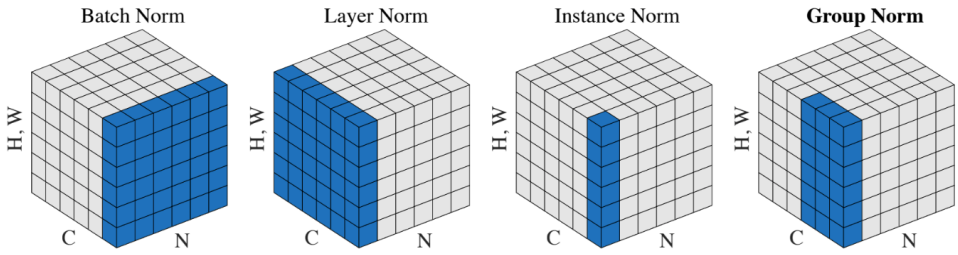


Figure 2.21: Comparison of feature normalization techniques [10]. Each of thesubplots shows a feature map tensor, where N denotes the batch axis, C is the channel axis and (H, W) are the spatial axes.

2.2.9 Regularization

Machine learning in general aims to create such an algorithm that would perform well not only on training data but more importantly it will perform well on unseen data. The error of an algorithm on unseen data is called *test error*. Many strategies aim to reduce test error and are collectively known as *regularization*.

A technique that penalizes the size of a weight vector $\boldsymbol{w}$ is called ***L2 regularization***. This regularization adds term $\lambda \|\boldsymbol{w}\|_2^2$ to the loss function, where λ is a parameter that controls the importance of regularization compared to other elements of the loss function. Very similar to $L2$ regularization is ***L1 regularization*** that penalizes sum of absolute values of weight's components, i.e. $\lambda \|\boldsymbol{w}\|_1$, where λ is once again a hyperparameter controlling the importance of this regularization.

To prevent overfitting of an algorithm to the training data, we often split the data available to *training set* and *validation set*. Once this is done, we train the algorithm with *training set* and validate it on *validation set*. We stop the algorithm once the validation loss starts to grow. This technique is often called ***early stopping***.

All of the regularizations aim to solve the problem of overfitting of neural networks to the training data. One of the possible ways to tackle this is an ensemble of neural networks with different configurations. However, this comes at the cost of having to train multiple models (which takes time) and maintain them (which might occupy quite some memory). ***Dropout*** [63] provides a computationally effective way of how to use a single model to simulate a large number of different network architectures. It allows training the ensemble of all possible sub-networks that can be formed by removing non-output units from an underlying base network [5]. During the training, dropout randomly *drops out* (removes) certain units. During each iteration of a neural network, a different binary mask $\boldsymbol{\mu}$ is sampled and applied to all hidden and input units of the neural network. The mask μ_i for each unit is sampled independently of the others. The probability of a mask for certain unit being 0 (which causes the unit to be removed for computing during this

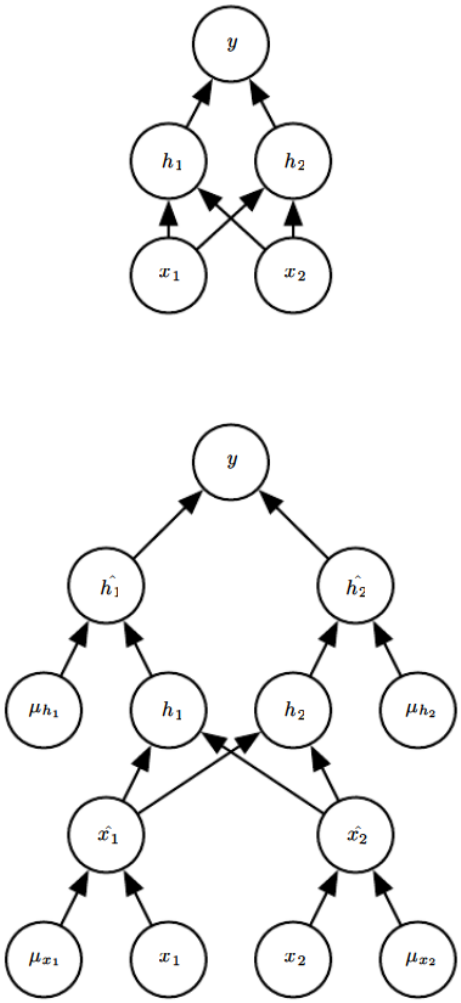


Figure 2.22: An example of dropout applied to a simple network architecture (top) [5]. In the bottom we can see a mask μ_i associated with each input and hidden unit. Each unit is multiplied with corresponding mask (either 0 or 1).

iteration) is a fixed hyperparameter. This hyperparameter can be different for hidden and input units. This is illustrated in Figure 2.22.

Another thing that can help the network to generalize better is to train it on a larger amount of data. However, in practice, we have only a limited number of data. One way to overcome this problem is to perform various transformations to the original data. This is called ***dataset augmentation***. For example, in the case of image classification, we might perform a horizontal flip of the image, apply small rotations, translate the image a few pixels or inject noise.

2.2.10 Architectures

In this section I will go through most common architectures of convolutional neural networks starting from LeNet-5 in section 2.2.10 and going all the way to more recent ResNet in section 2.2.10.

LeNet-5

LeNet-5 is a pioneering 7-layer convolutional neural network by LeCun in 1998 [2]. It was designed to recognize hand-written numbers of 32×32 gray-scale images.

The architecture of LeNet is shown in Figure 2.23. It consists of three 5×5 convolutional layers, two 2×2 pooling layers with stride 2 and two fully-connected layers at the end of the network.

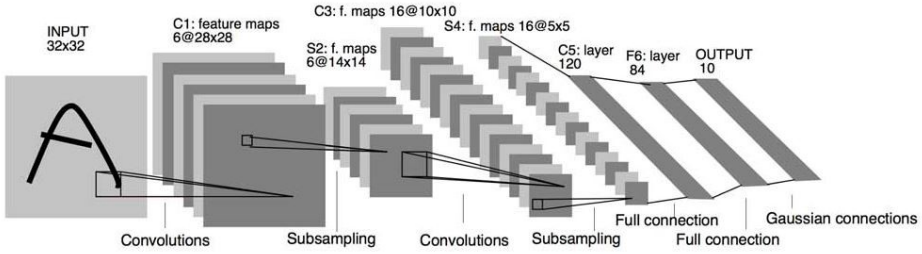


Figure 2.23: An architecture of LeNet-5 [2].

AlexNet

AlexNet [64] was a huge breakthrough back in 2012 when it outperformed all the prior competitors and reduced the top-5 loss from 26% to 15.3% in the computer vision challenge *imagenet* [65, 66]. The architecture was similar to LeNet-5, but it was deeper, with more filters and with stacked convolutional layers. The convolutions were of size 11×11, 5×5 and 3×3. Max pooling was used as a subsampling layer. It further applied data augmentation, used ReLU activations, and was trained with SGD with momentum. The detail of this architecture can be seen in Figure 2.24.

VGGNet

VGGNet [67] is one of the most appealing neural networks because of its simple and uniform architecture that can be seen in Figure 2.25. There are two basic rules:

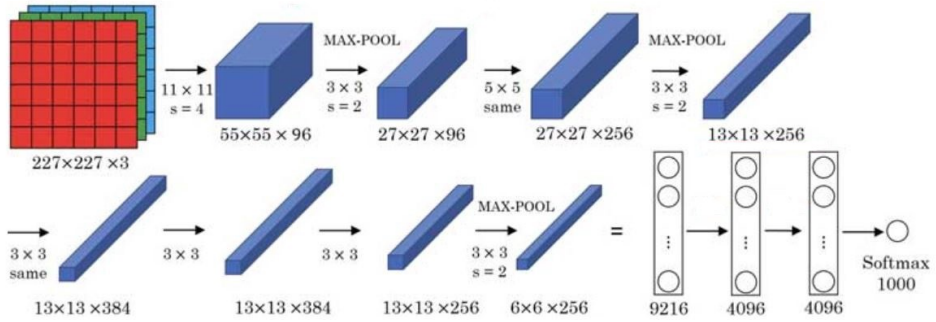


Figure 2.24: Architecture of AlexNet [11].

- Every convolutional layer has a kernel of size 3×3, stride 1, and zero padding 1. This means that the spatial size of the input is preserved.
- Every max pooling layer has a window of size 2×2 and stride 2. This results in halving the size with every such layer.

On top of convolutional and max-pooling layers are three fully connected layers. These layers have the majority (123M) of the total number parameters (138M). This network is commonly used for feature extraction from images.

Figure 2.25: Arichitecture of VGGNet [12].

GoogLeNet/Inception

GoogleNet [13] (also known as *Inception v1*) was the winner of imagenet competition in 2014 when it achieved a top-5 error of 6.67% which is near the human-level performance. In [13], authors proposed a novel concept dubbed *inception module* shown in Figure 2.26 that used smaller convolutions and allowed the network to have 4 million of parameters only while having 22 layers.

It uses batch normalization (2.2.8), image distortions, and is trained with RMSProp. The architecture of the whole network can be seen in Figure 2.27.

ResNet

Residual Neural Network [14] (or *ResNet* for short) introduces a novel concept of so-called *skip connections* shown in Figure 2.28 and uses batch normalization

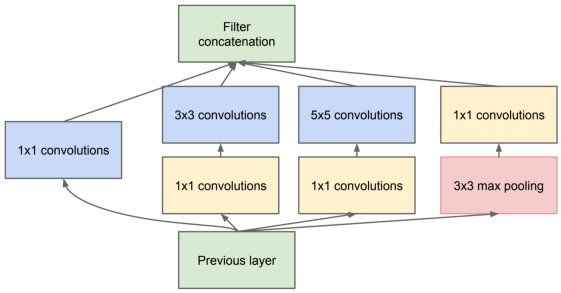


Figure 2.26: Inception module with dimensionality reduction [13].

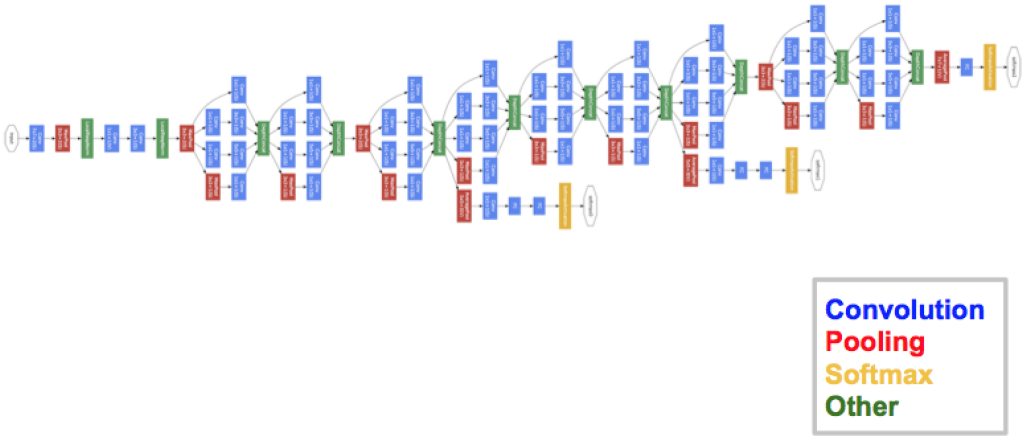


Figure 2.27: Architecture of GoogLeNet [12].

heavily. Thanks to this, the authors were able to train a neural network that consisted of 152 layers while the network was still less complex than VGGNet. It achieved top-5 error of 3.57% which is better than human-level performance. The architecture of ResNet is shown in Figure 2.29.

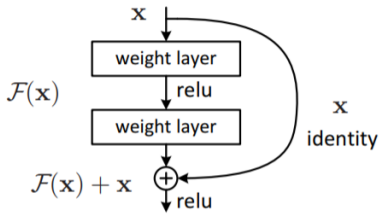


Figure 2.28: Residual block [14].

U-Net

U-Net is a convolutional neural network that was developed for biomedical image segmentation [15]. The architecture of U-Net is *fully convolutional*, which means that it uses only convolutional and no fully-connected layers.

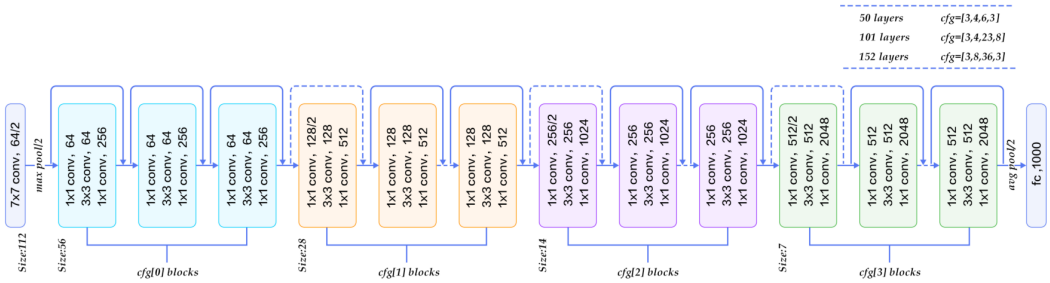


Figure 2.29: Architecture of ResNet [12].

The network consists of two paths:

1. *Contraction path* (also called encoder)
   The architecture of contraction path is a traditional one with convolutional layers followed by ReLU activations and max-pooling layers. In this path, the spatial information is reduced while the feature information (the number of channels) is increased. That is the reason why this path is also called a *encoder* since it encodes the information from the input.

2. *Expanding path* (decoder)
   The expanding path is symmetric to the contraction path with max pooling layers replaced by *transposed convolutions*. Transposed convolution is a technique used for upsampling that has trainable parameters. The expanding path concatenates the features obtained with convolutions and transposed convolutions in this path with high-resolution features from the contraction path as it can be seen in Figure 2.30.

The architecture of U-Net can is shown in Figure 2.30.

2.2.11 Multi-Task Learning

Multi-Task learning (MTL) is an inductive transfer mechanism whose principle goal is to improve generalization performance. MTL improves generalization by leveraging the domain-specific information contained in the training signals of the related task. This is done by training tasks in parallel while using a shared representation. In Figure 2.31 you can see a common form of MTL, where different tasks (that predict $\mathbf{y}^{(i)}$ given $\mathbf{x}$) share the input $\mathbf{x}$ and along with this, they share part of the computation network as well. This way, the parameters of a model can be divided into two main parts [5]:

1. *Task-specific parameters*. These parameters benefit from the samples of their task only.

2. *Generic parameters* are shared across all the tasks and benefit from the pooled data of all the tasks.

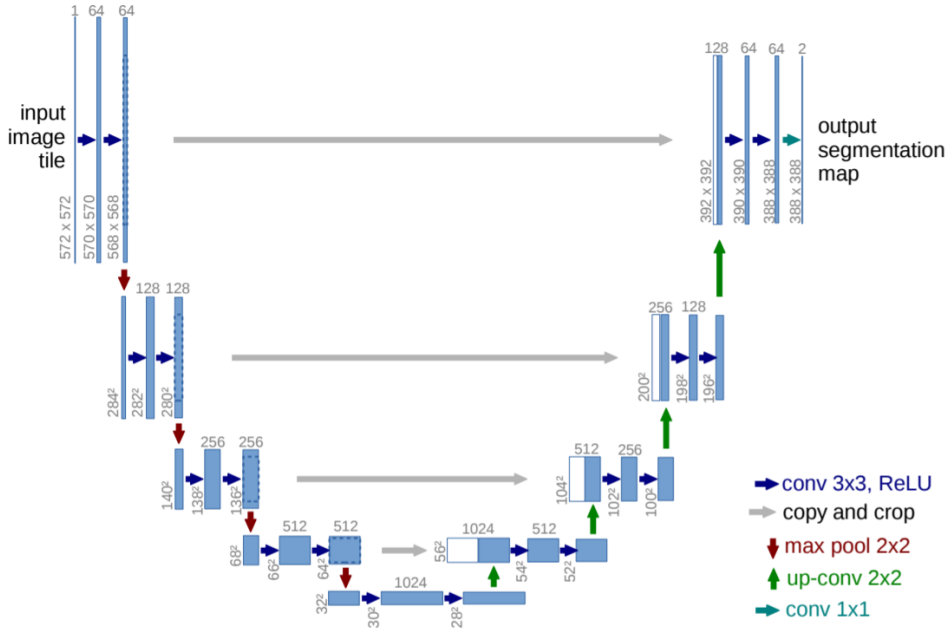
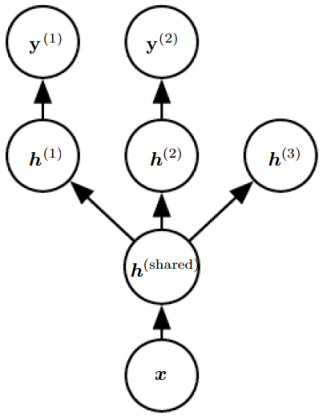


Figure 2.30: Architecture of U-Net [15].



.

Figure 2.31: An example of common MTL setup [5]. Generic parameters that are shared by all the tasks are here denoted as $\mathbf{h}^{(shared)}$.

Let us show an example. Imagine a neural network with the primary task of object detection. The secondary tasks used can be e.g., scene classification task or depth estimation.

The idea of Multi-Task Learning in the context of deep neural networks is extended in the work "Revisiting Multi-Task Learning with ROCK: a Deep Residual Auxiliary Block for Visual Detection" [16]. The authors propose a new architecture with a residual block where the features used for a secondary task are fused back with the features used for the primary task. This architecture is shown in Figure 2.33. As the authors say: "(residual

block) explicitly merges intermediate representations of the primary and auxiliary (secondary) tasks, making the latter ones have a real effect on the former in the forward pass, not just through shared feature learning" [16] - the difference is depicted in Figure 2.32. They show this architecture with the primary task of object detection that is trained with auxiliary tasks of scene class prediction, depth prediction, and surface normal prediction.

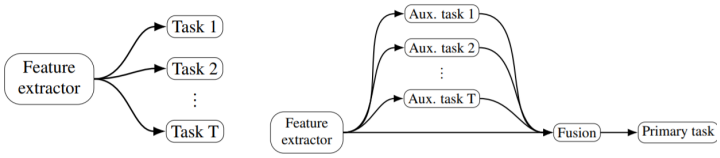


Figure 2.32: The difference between flat MTL (left) and MTL with ROCK (right) [16].

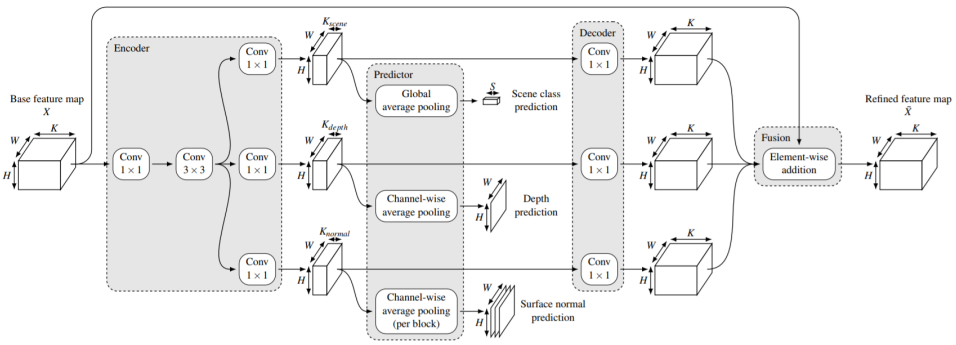


Figure 2.33: Architecture of the proposed residual block. Primary task of object detection that is trained with auxiliary tasks of scene class prediction, depth prediction and surface normal prediction [16]. The auxiliary tasks share two convolutional layers in encoder.

2.2.12 Transfer learning

Transfer learning is a commonly used technique in deep learning. It refers to the situation where what has been learned in one setting (e.g., a network trained on dataset 1) is exploited to improve generalization in another setting (e.g., performing the same task but on a different dataset) [5].

It is common to use some deep network that is pretrained on some large dataset (e.g., on ImageNet with 1000 classes) for classification. These pretrained networks provide a good starting point for further training since they were trained on a large dataset with many classes and therefore effectively extract features from the input images in order to classify well.

In general, the transfer learning provides a shortcut that can save training

time and lead to better results. Models that are initialized with some pretrained network often converge faster and achieve better results.

There are two possible ways of how to use some pretrained network:

1. use pretrained network as a feature extractor
2. retrain the complete network

In the first approach, the pretrained network is used as a feature extractor. In the classification task, this is done by removing the final layer of the original network that is used for classification and replacing it with a new one that has n outputs, where n is the number of classes. In the case of a VGG network, this would mean that we get a 4096-D vector after removing the last layer and use this vector as an input to a new final classification layer. During this setup, only the weights of the final layer are changed, and the rest of the network serves just as a feature extractor.

The second approach also replaces the final layer with a new one, but in contrast to the previous approach, it fine-tunes the weights in the whole network (or at least a significant part of it). The motivation behind this approach is in the observation that the early layers of a CNN contain general features such as edges and colors and can, therefore, be used for almost any task.

In both approaches, it is a common practice to use a smaller learning rate since the general features are already learned, and the network only needs to be finetuned.

2.3 Image data augmentation

Data augmentation is a technique that is used to enlarge the size of a dataset by creating a modified version of images from the dataset. Using image data augmentation often results in better performance and better generalization of the trained models since it is a common knowledge that in general deep learning models achieve better results when trained on more data.

In general, these methods increase the variability of the data in the final augmented dataset when compared to the original one. The methods based on image processing do not add any new information to the dataset.

The image data augmentation involves creating of transformed images from the original dataset. In general, the dataset augmentation methods can be divided into two classes:

1. *Offline*: In offline dataset augmentation, the various augmentation methods are performed on the images from the training split of the given dataset and are added to them. This results in a larger dataset.

2. *Online*: In online dataset augmentation, the augmentation methods are performed on the fly. This means that random augmentations are applied to the image during loading and are not stored.

The advantage of the offline method is that the transformations to the images in the training set are done only once and does not slow the data loading during the training. On the contrary, the online method requires that the image transformation is performed every time that the image is loaded and therefore leads to increased complexity of image loading.

On the other hand, more samples are created with the online method since the transformations on the image are different in each iteration.

The most common image transformations are:

- *Translation*: Each pixel is moved in the same direction. In the case that this transformation is applied to the whole image, it happens that some pixels get out of the image and some parts of the image that needs to be filled with a new value. In the case of pure horizontal/vertical shift, the pixels on the border can be copied to the created holes. If the translation is performed in both horizontal and vertical direction, one of the possible solutions is to rescale the valid pixels of the image to the size of the original image.
- *Horizontal and vertical flip*: Image is flipped either horizontally or vertically. The horizontal flip makes sense in almost every situation, but it does not hold for the vertical flip where one needs to pay attention. It does not, for example, make much sense to flip an image of a person vertically.
- *Rotation*: All pixels are rotated by the same random value. This once again results in a situation when some pixels get out of the image, and some parts of the image are empty. This can be tackled by nearest-neighbor fill.
- *Brightness change*: The brightness change is intended to allow the model to better generalize across images with various lighting conditions. All the pixels are either darkened or brightened.
- *Zoom*: An image is randomly either zoomed-in or zoomed-out, and the new pixel values are either added (e.g., with nearest-neighbor fill) or interpolated.
- *Gaussian Noise*: In Gaussian Noise augmentation, one adds a randomly sampled value from a Gaussian distribution to each pixel in the image. This is intended to tackle the problem of network overfitting to high-frequency features.

Each of the augmentation techniques mentioned above can either be used alone, or they can be combined together.

Recently, a new type of neural networks called Generative Adversarial Networks was proposed, and it can be used for image data augmentation as well. This technique is introduced in the following section.

2.4 Generative Adversarial Networks

Generative Adversarial Network (GAN) belongs to the class of generative algorithms 2.1.5 and thus can be used for data generation. One of the augmentation methods proposed in this thesis is based on such a network. Authors of [1] define GANs as follows:

> GAN is a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a *generative model* G that captures the data distribution, and a *discriminative model* D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a *minimax two-player game*. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to $1/2$ everywhere [1].

As written above, this framework consists of two neural networks: discriminator D and a generator G.

- *Discriminator* is a discriminative model that learns to determine whether the input data come from the real data distribution or whether the data were generated by a generator.
- The opponent to the discriminator is called *generator* and it is a kind of a generative model. The goal of the generator is to generate data that would resemble the real data as much as possible. The generator, therefore, learns the distribution of training samples.

The architecture of this framework is shown in Figure 2.34

2.4.1 Vanilla GANs

In the original paper by Goodfellow [1] both networks, discriminator D and generator G, are multilayer perceptrons. I will show in the following sections that these networks are not restricted to be MLP only but can be convolutional neural networks witch yields much better results compared to the basic setup with MLP.

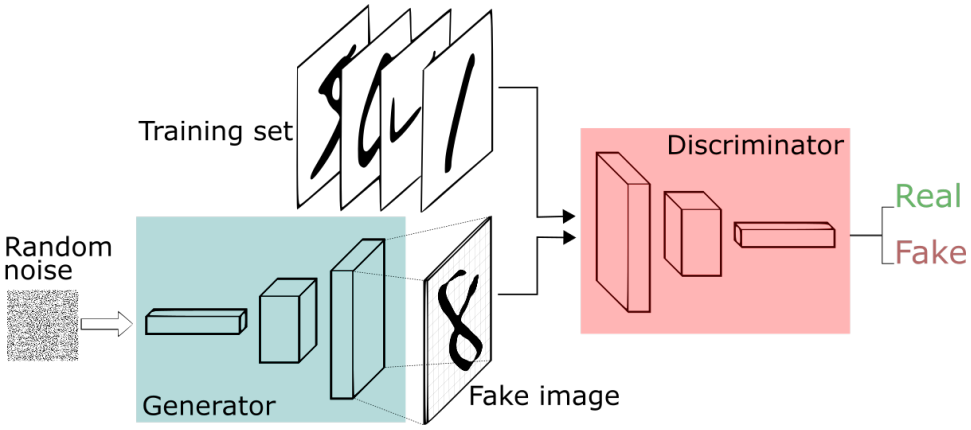


Figure 2.34: Architecture of GAN for generating hand-written digits [17].

In order to learn generator's distribution p_g over data $\mathbf{x}$, let us define a prior on input noise variables $p_z(z)$ that represents a mapping to data space as $G(\mathbf{z}, \theta_g)$. Differentiable function G is a multilayer perceptron with parameters θ_g that learns to map from the space of random input noise variables $\mathbf{z}$ to the space of target data distribution and tries to match this distribution as accurately as possible.

The discriminator is modeled as an MLP $D(\mathbf{x}, \theta_d)$ that outputs a single scalar representing the probability that the input $\mathbf{x}$ came from the real data rather then from p_g [1].

We train these two networks to compete against each other. The goal of the discriminator D is to assign the correct labels to both real training examples (to which it should assign 1) and to samples generated by G (to which it should assign 0). The generator is simultaneously trained to fool the discriminator in believing that all the generated samples come from the real training distribution. The generator wants the discriminator to output 1 for all the generated samples, i.e. $D\left(G(z)\right) = 1$. This corresponds to minimizing $\log\left(1 - D\left(G(z)\right)\right)$. In other words, D and G play a two-player minimax game with the values function $V(G, D)$ 2.48.

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{ data }}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}}(\boldsymbol{z})[\log(1 - D(G(\boldsymbol{z})))] \tag{2.48}$$

In practice, using the equation 2.48 doesn't necessarily have to provide a good gradient to G, and it thus doesn't have to learn well. This can be seen early in the training when the generator is poor in generating believable data and discriminator, therefore, rejects the generated samples with high confidence. This results in a saturation of the term $\log(1 - D(G(\boldsymbol{z})))$. Therefore it is better to rather maximize $\log D(G(z))$ which results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in the training [1].

We may rewrite the loss function 2.48 as an integral over inputs x as in

equation 2.49

$$L(D,G) = \int_x \left(p_r(x)\log(D(x)) + p_g(x)\log(1-D(x))\right) dx \tag{2.49}$$

Having this form, let us label $\tilde{x} = D(x), A = p_r(x), B = p_g(x)$, where p_g is the distribution of the generator and p_r is the distribution of the training data. Using this notation we get 2.50

$$f(\tilde{x}) = A\log\tilde{x} + B\log(1-\tilde{x}) \tag{2.50}$$

The derivative of the function 2.50 with respect to $\tilde{x}$ is 2.51

$$\frac{\partial f(\tilde{x})}{\mathrm{d}\tilde{x}} = \frac{1}{\ln 10}\frac{A-(A+B)\tilde{x}}{\tilde{x}(1-\tilde{x})} \tag{2.51}$$

Thus, we obtain an optimal value by setting the result of 2.51 equal to zero

$$D^*(x) = \tilde{x}^* = \frac{A}{A+B} = \frac{p_r(x)}{p_r(x)+p_g(x)} \in [0;1] \tag{2.52}$$

In the case that the generator is trained to its optimal, its distribution p_g gets very close to the real distribution p_r. Therefore, in the case when $p_g = p_r$ we get for the optimal value of the discriminator D

$$D^*(x) = \frac{p_r(x)}{p_r(x)+p_r(x)} = \frac{p_r(x)}{2p_r(x)} = \frac{1}{2} \tag{2.53}$$

Now we can see that for G and D at their optimal values we have

- $p_g = p_r$
- $D^*(x) = \frac{1}{2}$

and we can show that the loss function 2.49 evaluates for these values to

$$\begin{aligned} L(D,G) &= \int_x \left(p_r(x)\log\left(D^*(x)\right) + p_g(x)\log\left(1-D^*(x)\right)\right) dx \\ &= \log\frac{1}{2}\int_x p_r(x)dx + \log\frac{1}{2}\int_x p_g(x)dx = 2\log\frac{1}{2} = -2\log 2 \end{aligned} \tag{2.54}$$

In order to better understand what does the loss function represents, there needs to be shown two metrics of similarity first. The first metric is called ***Kullback-Leibler (KL) divergence*** and it measures how much does one probability distribution p diverge from the other distribution q. The KL divergence is defined as

$$D_{KL}(p\|q) = \int_x p(x)\log\frac{p(x)}{q(x)}dx \tag{2.55}$$

The minimum of 2.55 is at 0 and is achieved when the probability distributions p and q are equal, i.e. $p(x) = q(x) \forall x$. The disadvantage of KL divergence is that it is *assymetric*.

Jensen-Shannon (JS) divergence is yet another similarity measure between two probability distributions p and q and is bounded by $[0, 1]$. It is defined as

$$D_{JS}(p\|q) = \frac{1}{2} D_{KL}\left(p\|\frac{p+q}{2}\right) + \frac{1}{2} D_{KL}\left(q\|\frac{p+q}{2}\right) \tag{2.56}$$

JS divergence is symmetric and more smooth then KL divergence. We can now write the JS difference between p_r and p_g

$$\begin{aligned} D_{JS}\left(\boldsymbol{p}_r\|\boldsymbol{p}_g\right) &= \frac{1}{2} D_{KL}\left(p_r\|\frac{p_r+p_g}{2}\right) + \frac{1}{2} D_{KL}\left(p_g\|\frac{p_r+p_g}{2}\right) = \\ &= \frac{1}{2}\left(\log 2 + \int_x p_r(x) \log \frac{p_r(x)}{p_r(x)+p_g(x)} dx\right) \\ &+ \frac{1}{2}\left(\log 2 + \int_x p_g(x) \log \frac{p_r(x)}{p_r(x)+p_g(x)} dx\right) = \\ &= \frac{1}{2}\left(\log 4 + L\left(G, D^*\right)\right) \end{aligned} \tag{2.57}$$

Thus:

$$L\left(G, D^*\right) = 2 D_{JS}\left(p_r\|p_g\right) - 2\log 2 \tag{2.58}$$

Essentially the loss function 2.58 quantifies the similarity between the generator data distribution p_g and the real sample distribution p_r by JS divergence when the discriminator is optimal. The optimal generator G^* that replicates the real data distribution leads to the minimum value of zero Jensen-Shannon divergence and thus to

$$L(G^*, D^*) = -2\log 2 \tag{2.59}$$

In the setup from original paper [1], the authors did experiments with MNIST, TFD, and CIFAR-10. Examples of results of experiments with these datasets can be seen in Figure 2.35

The training procedure proposed in [1] can be seen in 2.36.

There are quite a few problems of GANs in this basic setup. To begin with, it is hard to achieve Nash equilibrium which is a solution concept of a non-cooperative game involving two or more players in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only their own strategy [68]. In GANs, each model updates its cost independently with no respect to another player in the game and updating the gradient of both models concurrently cannot guarantee convergence.

Another problem is in the fact that the dimensions of many real-world datasets, as represented by the data distribution p_r, only appear to be

Figure 2.35: Examples of generated samples. The rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. In a) are examples from MNIST dataset, in b) examples from TFD dataset, in c) from CIFAR-10 trained with fully-connected network and in d) examples from CIFAR-10 trained with a convolutional network.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 2.36: GAN training procedure proposed in [1].

artificially high. They have been found to concentrate in a lower dimensional manifold. In example of the real-world images, once the theme or the contained object is fixed, the images have a lot of restrictions to follow, i.e., a cat should have two ears and a tail, and a human face contains two eyes with nose between them and a mouth below the nose, etc. These restrictions keep images away from the possibility of having a high-dimensional free form. The generator distribution p_g over data lies in a low dimensional manifold, too. Whenever the generator is asked to a much larger image like 64×64 with 3 channels given a small dimension, such as 128, noise variable input, the distribution of colors over these $64 \times 64 \times 3 = 4096$ pixels has been defined by the small 128-dimension random number vector and can hardly fill up the whole high dimensional space. Because these two distributions rest in low dimensional manifolds, they are almost certainly going to be disjoint as seen in the figure in Figure 2.37 [18].

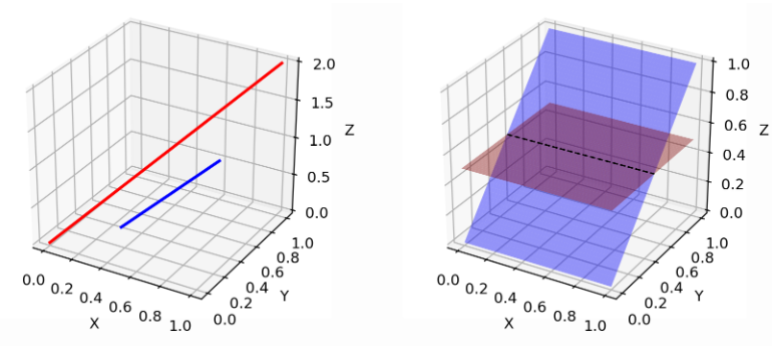


Figure 2.37: Two examples of low dimensional manifolds in 3D that can hardly have overlaps [18].

Then we have a typical problem with neural networks – *vanishing gradient*. In the case of the perfect discriminator, we are guaranteed with discriminator outputting 1 for all real samples and 0 for the generated ones. In this case, the loss function falls to zero, and there is no gradient to update with. Therefore, we face a dilemma:

- If we have a *poor discriminator*, the generator has *no accurate feedback*
- On the other hand, if we have a *very good discriminator*, the *gradient of the loss function goes to* 0, and this results in slow or even jammed learning.

The common problem in the training of GANs is called *mode collapse*. This is the state of the generator when it collapses to a setting where it *always produces the same outputs*. For example, a generator might be able to reproduce one of the training samples with 100% accuracy and therefore be able to fool the discriminator but would fail to capture the complex distribution of the complete dataset.

Another of the problems is one of the *evaluation metrics*. GAN objective function does not give good feedback about when to stop as can be seen in Figure 2.38. You can see that the objective function does not change much, but the difference in the quality of generator outputs can be quite huge.

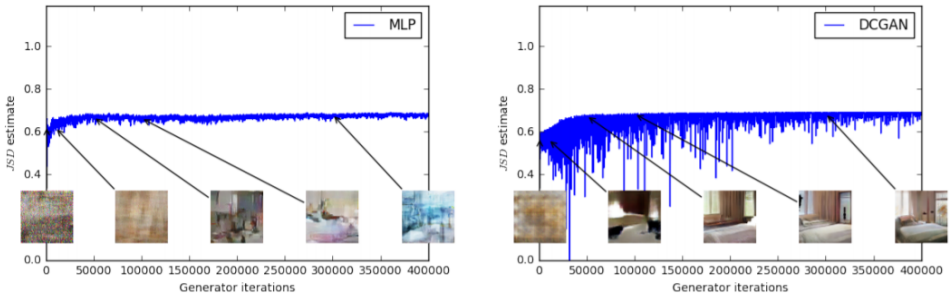


Figure 2.38: JS divergence for MLP generator (left) and DCGAN generator (right). In the example with DCGAN, you can see that the produced samples get better over time, but the JS divergence increases or stays constant [19].

Improvements to GAN training

There are various modifications [69, 70, 18] that help to stabilize and improve GAN training.

Feature matching. *Feature matching* [69] is a technique that suggests to optimize the discriminator to inspect whether the generator output matches expected statistics of the real samples [18]. In this case, the loss function is in the form $\left\|\mathbb{E}_{x\sim p_r} f(x) - \mathbb{E}_{z\sim p_z(z)} f\left(G(z)\right)\right\|_2^2$, where $f(x)$ can be any computation of statistics of features. One common way in image generation is to match features of certain intermediate layer of a pretrained network (e.g. VGG). Another approach is to try to match directly features obtained from intermediate layer of discriminator.

Historical averaging. *Historical averaging* [69] enforces the parameters of both models not to change their waights too dramatically over time. This is done by adding a term $\left\|\Theta - \frac{1}{t}\sum_{i=1}^{t}\Theta_i\right\|^2$ to loss function of both networks where θ are the parameters of current model and Θ_i is how the parameters were configured in a previous time step i.

One sided label smoothing. *One sided label smoothing* [69] suggests to use softened values for target class, such as 0.9 (for real samples) and 0.1 (for generated ones) instead of 1 and 0.

Adding noises. As it was stated among problems, the distributions p_r and p_g are most probably disjoint in a high dimensional space and it causes the problem of a vanishing gradient. In [70] they propose to add continuous noise to the input of the discriminator in order to get a higher chance for these two probability distributions to have overlaps.

Better metric. Since one of the biggest problems of training of GANs is in the fact that vanilla GANs measure the JS divergence between p_r and p_g distributions which fails to provide a meaningful values in the case of disjoint distributions, authors of [70] propose to use another metric such as Wasserstein distance (will be discussed in 2.4.2).

Conditional GAN

We can get a conditional GAN [71] by a simple modification. It is constructed by simply feeding the data, denoted as y, that we wish to condition on to both the generator and the discriminator. You can see the structure proposed in [20] in the Figure 2.39. We have noise input z and data y that we want to condition on as inputs to the generator. The same data y will then be fed to the discriminator.

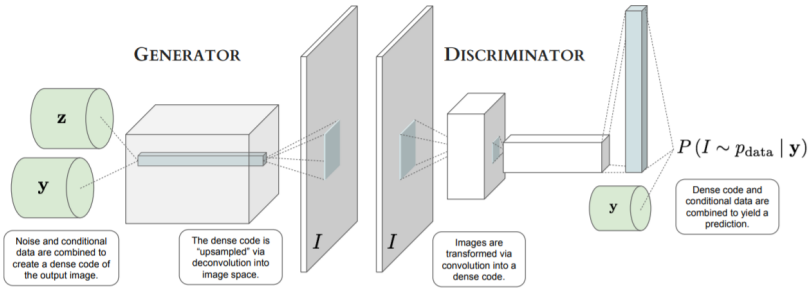


Figure 2.39: Structure of conditional GAN proposed in [20].

DCGAN

DCGAN [21] is an all-convolutional network that eliminates fully-connected layers on top of the convolutional network. The architecture can be seen in Figure 2.40. In all but the last layers of the generator and in all but the first layers of the discriminator the batch normalization (sec. 2.2.8) is used. As activations, the generator uses ReLU (except for the last layer where there is sigmoid activation), and discriminator uses Leaky ReLU.

Walking on the learned manifold can usually tell us about signs of memorization (if there are sharp transitions). If walking in this latent space results in semantic changes to the image generations (such as objects being added and removed), we can reason that the model has learned relevant and interesting

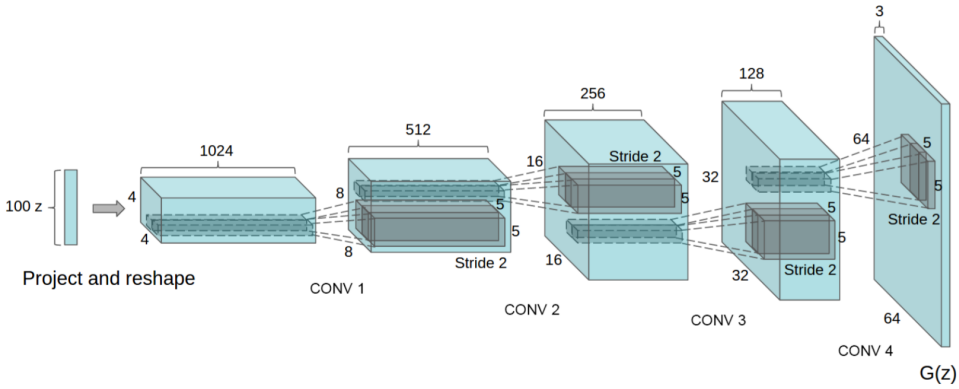


Figure 2.40: Architecture of DCGAN [21].

representations [21]. An example from [21] of such a manifold walk is shown in figure 2.41.

Figure 2.41: Wlking the learnt manifold of LSUN bedroom dataset [21].

Authors of [72] demonstrated that simple arithmetic operations reveal rich linear structure in learned space. They showed a canonical example that the vector("King") - vector("Man") + vector("Woman") resulted in a vector whose nearest neighbor was the vector for "Queen" which follows the basic human intuition. The authors of [21] investigated whether this linear arithmetic holds in the Z space (random input noise) of the generator. They observed that experiments with only one sample per concept were unstable. However, a simple averaging of three vectors from the Z space produced results that were stable and obeyed the intuition and arithmetic. An example of such arithmetic can be seen in Figure 2.42.

2.4.2 Wasserstein GAN

Before diving into Wasserstein GANs, it is important to introduce the *Wasserstein distance first*. It is yet another measure between two probability distributions. It is also known as the *Earth Mover's distance* (*EM* for short).

Wasserstein distance denotes how much "mass" must be transported from x to y in order to transform the distribution p_r into the distribution p_g. The EM distance then is the "cost" of the optimal transport plan and can be formulated as

$$W\left(p_r, p_g\right) = \inf_{\gamma \in \Pi\left(p_r, p_g\right)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{2.60}$$

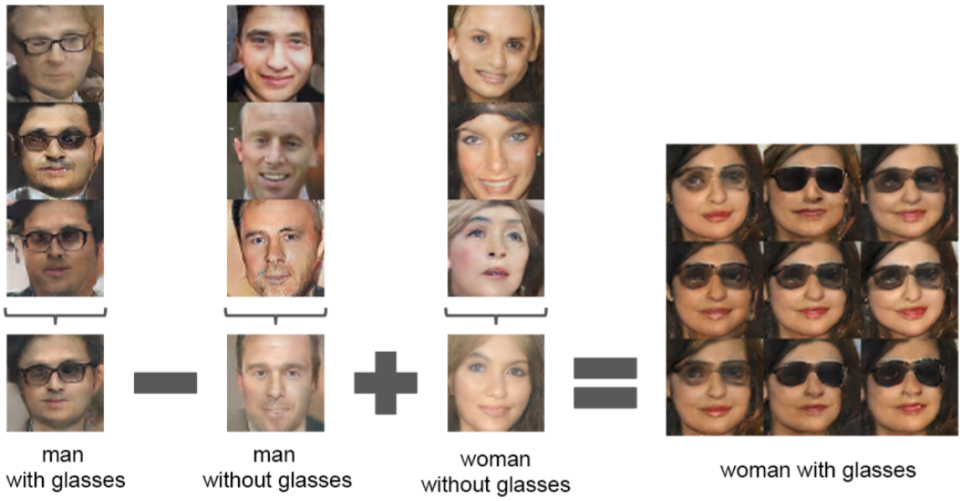


Figure 2.42: An example of vector arithmetic with human faces [21]. Three vectors for each concept were averaged.

where $\Pi(p_r, p_g)$ denotes the set of all possible joint probability distributions $\gamma(x, y)$ between p_r and p_g. As one might guess, $\gamma(x, y)$ indicates how much "mass" must be transported from x to y in order to transform the distribution p_r into the distribution p_g [19].

Compared to the already introduced KL and JS divergences, the Wasserstein distance behaves better in the case of non-overlapping distributions and provides a smooth and meaningful representation of the distance in-between. This comparison can be demonstrated on a small example showed in 2.43 [18]. There are two distributions which you can see in Figure 2.43

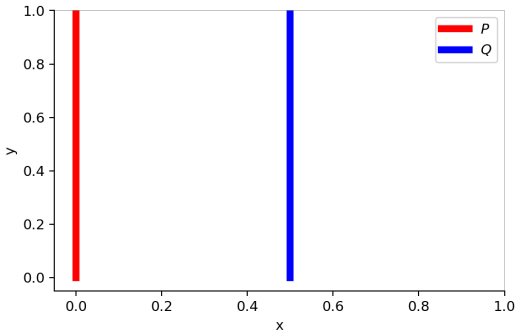


Figure 2.43: An example of distributions [18].

- In red we have distribution p which has x coordinate fixed at 0 and y coordinate is sampled uniformly from the interval from 0 to 1
- In blue we have the second distribution q that is parametrized by θ that denotes its x coordinate that lies in the range $[0, 1]$ and its y coordinate is again sampled uniformly from 0 to 1.

Let us compare different metrics on the example showed in 2.43. In the case when $\theta \neq 0$

- For KL divergence we obtain:

$$\begin{aligned} D_{KL}(p\|q) &= \sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty \\ D_{KL}(p\|q) &= \sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty \end{aligned} \tag{2.61}$$

- For the JS divergence we get:

$$D_{JS}(p,q) = \frac{1}{2}\left(\sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{1/2} + \sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{1/2}\right) = \log 2 \tag{2.62}$$

- However, for EM (Wasserstein) distance W we get:

$$W(p,q) = |\theta| \tag{2.63}$$

In the case when $\theta = 0$

- For KL and JS divergence:

$$D_{KL}(p\|q) = D_{KL}(q\|p) = D_{JS}(p,q) = 0 \tag{2.64}$$

- For EM (Wasserstein) distance:

$$W(p,q) = 0 = |\theta| \tag{2.65}$$

From this, we can see that D_{KL} is ∞ when the two distributions are disjoint. The comparison of EM distance and JS divergence is in Figure 2.43. In the case of JS divergence D_{JS}, there is a sudden jump from and to $\log 2$, and it is not differentiable at 0. On the contrary, the Wasserstein metric provides a smooth measure.

A ***Wasserstein GAN*** (WGAN for short) is a form of GAN that minimizes a reasonable and efficient approximation to the EM distance. The infimum in 2.60 is highly intractable. On contrary, the Kantorovich-Rubinstein duality tells that

$$W(p_r, p_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim p_r}[f(x)] - \mathbb{E}_{x\sim p_\theta}[f(x)] \tag{2.66}$$

where the supremum is over the all 1-Lipschitz functions $f : \mathcal{X} \to \mathbb{R}$. In the case that we replace $\|f\|_L \leq 1$ for $\|f\|_L \leq K$ (that corresponds to K-Lipschitz functions for some constant K) we end up with $K \cdot W(p_r, p_g)$. Therefore, if we

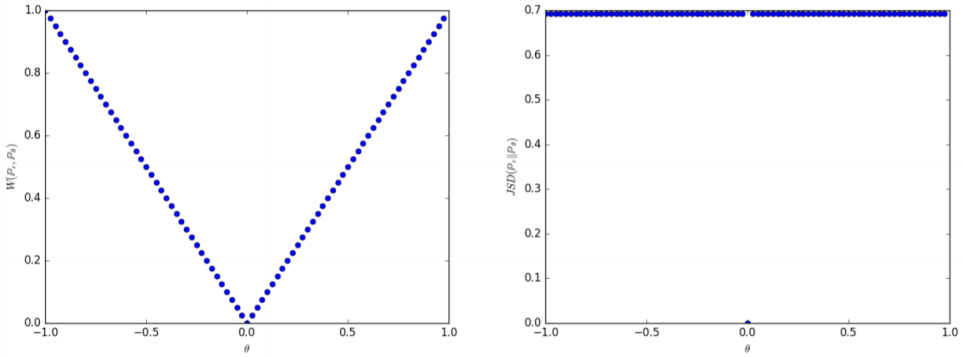


Figure 2.44: Comparison of the EM distance (left) and the JS divergence (right) [19].

have a parameterized family of functions $\{f_w\}_{w\in\mathcal{W}}$ that are all K-Lipschitz for some constant K, we can consider to solve the problem

$$\max_{w\in\mathcal{W}} \mathbb{E}_{x\sim\mathbb{P}_r}\left[f_w(x)\right] - \mathbb{E}_{z\sim p(z)}\left[f_w\left(g_\theta(z)\right)\right] \tag{2.67}$$

and if the supremum in 2.66 is attended for some $w \in \mathcal{W}$, this process fill yield to a calculation of a Wasserstein distance $W(p_r, p_\theta)$ (up to a multiplicative constant) [19]. The differentiation of $W(p_r, p_\theta)$ by backpropagating through equation 2.66 gives us (up to a constant)

$$\mathbb{E}_{z\sim p(z)}\left[\nabla_\theta f_w\left(g_\theta(z)\right)\right] \tag{2.68}$$

Therefore we have a loss function in the form

$$L\left(p_r, p_g\right) = W\left(p_r, p_g\right) = \max_{w\in W} \mathbb{E}_{x\sim p_r}\left[f_w(x)\right] - \mathbb{E}_{z\sim p_r(z)}\left[f_w\left(g_\theta(z)\right)\right] \tag{2.69}$$

We should note a difference in the role of a discriminator known from the vanilla GAN. The "discriminator" in Wasserstein GAN does not aim to tell the real and fake samples apart, but it is trained to learn K-Lipschitz function to help to compute EM distance instead. Therefore, the former GAN's discriminator is now called *critic* in WGANs.

There arises one more problem of enforcing the K-Lipschitz continuity of function f_w during the training procedure. In the original paper [19] they propose a simple trick of clamping the weights w to a small window, e.g., to $[-0.01, 0.01]$, after every gradient update, which results in a compact space W leading to f_w with its upper and lower bound to preserve the Lipschitz continuity.

The original training procedure of WGAN training proposed in [19] is shown in 2.45. The fact that the EM distance is continuous and differentiable a.e. means that we can (and should) train the critic till optimality. The reason is simple: the more we train the critic, the more reliable gradient we get. In Figure 2.46, you can see the difference between GAN discriminator and WGAN

critic when trained to optimality. The discriminator learns very quickly to distinguish between fake and real, and as expected, provides no reliable gradient information. The critic, however, can't saturate, and converges to a linear function that gives remarkably clean gradients everywhere [19]. In contrary to training of GAN, we can see in Figure 2.47 that the Wasserstein distance correlates well with the sample quality.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.
while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 2.45: WGAN training procedure [19].

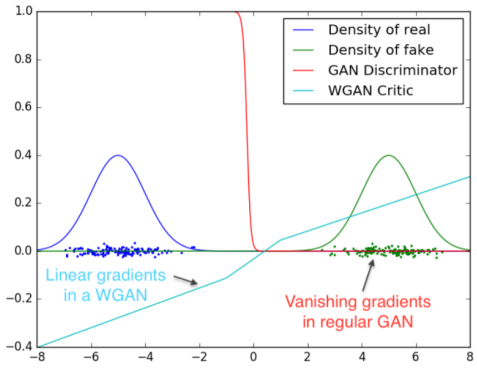


Figure 2.46: Discriminator of vanilla GAN saturates and results in vanishing gradients. On contrary, WGAN critic provides clean gradients [19].

The difference to the original GAN training procedure 2.36 is that:

- Use of RMSProp in WGAN training.
- In WGAN training, after every gradient update of the critic function, the weights are clamped to a fixed range $[-c, c]$ where c is a hyperparameter.

The advantages of WGAN over GAN include:

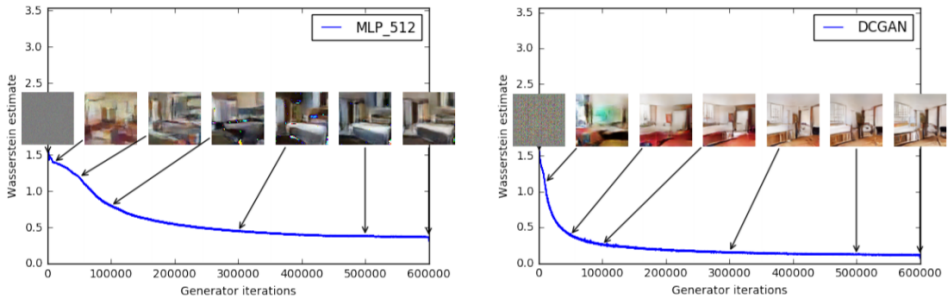


Figure 2.47: In the training curves of WGAN, we can see a clear correlation between sample quality and error value [19].

- Earth Mover's (Wasserstein) distance is continuous and differentiable, which results in the fact that we can (and should) train the critic till optimality.
- The WGAN training does not require maintaining a careful balance in training of the discriminator (critic) and the generator since the more we train the critic, the more reliable gradients we get.
- The mode collapse that is typical in GANs is also reduced.

Despite the many advantages of WGAN, there is also a problem that was already mentioned by the authors [19]. This problem is in the enforcing of K-Lipschitz continuity by weight clipping about which the authors say the following: "Weight clipping is a clearly terrible way to enforce a Lipschitz constraint". A possible replacement for the weight clipping is *gradient penalty* that will be introduced in the following section.

Gradient penalty

Gradient penalty [22] is an alternative way to enforce Lipschitz constraint. A differentiable 1-Lipschitz function is a function that has gradients with the norm at most 1 everywhere. Therefore, the authors of [22] propose to directly constrain the gradient norm of the critic's output with respect to its input.

The gradient penalty is enforced as a soft version of a constraint with a penalty on the gradient norm for random samples. The new objective can be seen in equation 2.70. The first two terms are just the same as in the original loss, and the remaining term is the new gradient penalty weighted by a penalty coefficient λ that is set to 10 in the original paper.

$$L = \mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim p_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim p_r}[D(\boldsymbol{x})] + \lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim p_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right] \tag{2.70}$$

The samples $\hat{\boldsymbol{x}}$ for computing the gradient penalty are sampled from the distribution $p_{\hat{x}}$ that is defined by uniform sampling along straight lines between pairs of points sampled from the real distribution p_r and generated points from generator distribution p_g.

The weight clipping biases the critic towards much simpler functions as you can see in the top row in Figure 2.48 . In the bottom row, you can see that training with gradient penalty does not suffer from this problem.

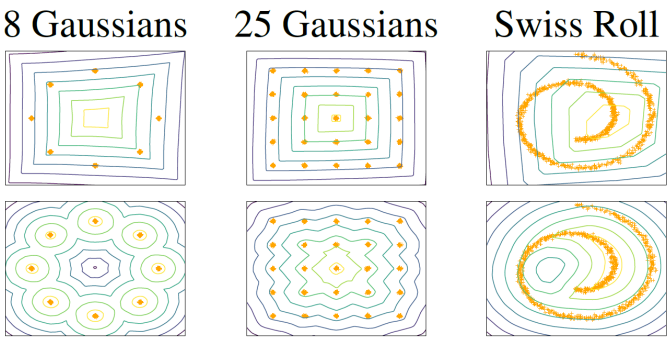


Figure 2.48: Comparison of value surfaces of WGAN critics trained to optimality on toy datasets using (top) weight clipping and (bottom) gradient penalty [22].

2.4.3 Other techniques and architectures

There are multiple other techniques and architectures that are used with Generative Adversarial networks. In this section, I will briefly describe a few important ones.

Image-to-image translation

In [73], the authors investigate conditional networks as a general-purpose solution to image-to-image translation problems. The advantage of these networks is that they do not only learn the mapping from input to output image, but they also learn a loss function that is used to train such networks.

Vanilla version of GANs 2.4.1 is trained to learn to map a random noise vector z to an output image y by generator $G : z \rightarrow y$. Conditional GANs 2.4.1 learn a mapping from observed input image x and random noise vector z, to output image y, i.e. $G : x, z \rightarrow y$. An example of image-to-image translation with the network named ***pix2pix*** is depicted in Figure 2.49. For the case of conditional GANs, the loss function becomes

$$\mathcal{L}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))] \tag{2.71}$$

The authors also propose to add L1 loss 2.26 to 2.71 as it forces the generator to reconstruct the ground truths and encourages less blurring compared to

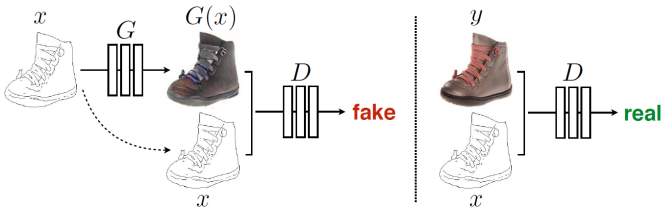


Figure 2.49: An example of a conditional GAN training to map edges to photo.

L2 loss used in [74]. This L1 loss has a form of

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}\left[\|y - G(x,z)\|_1\right] \tag{2.72}$$

Incorporating the L1 loss to the general conditional GAN loss we arrive to

$$G^* = \arg\min_C \max_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G) \tag{2.73}$$

An example of pix2pix trained to map from Google Maps to aerial photo and from aerial photo to Google Maps can be seen in Figure 2.50

Figure 2.50: Example of pix2pix trained to map from Google Maps to aerial photo (left) and from aerial photo to Google Maps (right).

Image inpainting. A task of filling missing pixels of an image is often known under the name *image inpainting* and can be seen as a part of the image-to-image translation. The main challenge in this task is to generate realistic and semantically plausible pixels for the missing regions. These newly synthesized pixels should be coherent with the existing ones. In the early works [75, 76], it was attempted to tackle this task by ideas similar to texture synthesis [77, 78], i.e. by matching and copying background patches into holes starting from low- to high-resolution or propagating from hole boundaries [23]. These

approaches may work well for stationary textures but usually fail for more difficult scenarios as inpainting to the natural images.

The first efforts to solve this task with deep neural networks were made by [79, 80], where the authors aimed to denoise the input and inpaint small regions in the image. In Context Encoders [74] they train a deep network for inpainting large holes first. More specifically, they train the network to fill a centered region of 64×64 pixels in a 128×128 pixels large image. As the objective function, they combine a standard GAN loss with L2 reconstruction loss. Further improvements were proposed in [81] by incorporating global and local discriminators. The job of the *global* discriminator is to assess that the whole image is coherent, while the goal of *local* discriminator enforces local consistency by focusing on a small area of the generated region only.

Authors of [23] propose a generative inpainting framework with the architecture showed in 2.51. They use a coarse-to-fine network architecture. The generator network takes an image with white pixels filled in the holes and a binary mask indicating the hole regions as input pairs, and outputs the final completed image, just as in [81]. They pair the input with a corresponding binary mask to handle holes with variable sizes, shapes and locations [23].

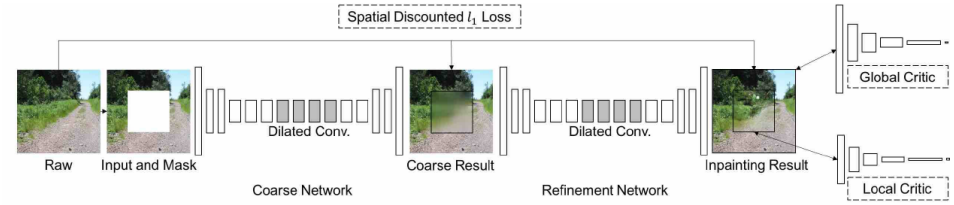


Figure 2.51: An architecture of generative inpainting framework [23].

In the task of image inpainting, the size of the receptive fields in the network should be sufficiently large [23]. Iizuka [81] proposes to use dilated convolution in order to tackle this. To further enlarge the receptive fields, the authors of [23] propose a two-stage coarse-to-fine architecture, where the first stage makes an initial coarse prediction and works as an input to the second stage that produces the final results. There is a difference in the losses that are used in the first (coarse) and second (refinement) stage. In the coarse stage, the network is trained with the reconstruction loss only, while the refinement stage adds to this loss a GAN loss. However, this network is trained mainly for the task of filling rectangular holes.

Cycle consistency

In work [24] they propose a novel approach to image-to-image translation, specifically to domain transfer, for learning to translate an image from some source domain X to a target domain Y *in absence of paired data*. The goal is therefore to learn a mapping $G : X \rightarrow Y$ such that the distribution of images from $G(X)$ would be indistinguishable from the distribution of images in Y

with the use of an adversarial loss. To deal with such an under-constrained problem, they propose to add an *inverse mapping* $F : Y \to X$ and to use so-called *cycle consistency* to enforce $F(G(x)) \approx X$ and $G(F(y)) \approx y$.

The model is described in Figure 2.52. As shown in (a), the model consists of two mapping functions X and Y that are associated with corresponding adversarial discriminators denoted as D_X and D_Y. D_Y encourages G to translate samples from X domain in such a way that they would be indistinguishable from samples from Y domain. The same holds for discriminator D_Y and mapping function F. In (b) and (c) is shown the idea of cycle consistency loss. In (b) you can see forward cycle-consistency when a sample $x \in X$ is mapped by function G as $\hat{y} = G(x)$ to Y domain and mapped back by function F as $\hat{x} = F(\hat{y})$. The cycle-consistency aim to arrive to the point when the $x \approx \hat{x}$. The cycle-consistency loss then penalizes the L1 difference of original sample x and the generated sample $\hat{x}$. The same holds for backward cycle-consistency for $y \in Y$ in (c). The overall cycle consistency is then given by

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x\ p_{data}(x)} \left[\|F(G(x)) - x\|_1\right] + \mathbb{E}_{y\ p_{data}(y)} \left[\|G(F(y)) - y\|_1\right] \tag{2.74}$$

The adversarial loss for the mapping $G : X \to Y$ is expressed as in equation 2.75 and vice versa for F.

$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)} \left[\log D_Y(y)\right] + \mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\log\left(1 - D_Y(G(x))\right)\right] \tag{2.75}$$

The overall loss function then adds typical GAN loss for both G and F mappings as in the form

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_{cyc}(G, F) \tag{2.76}$$

where λ is a parameter controlling the relative importance of the objective. The goal is then to solve the problem in the form

$$G^*, F^* = \arg \min_{G,F} \max_{D_x, D_Y} \mathcal{L}(G, F, D_X, D_Y) \tag{2.77}$$

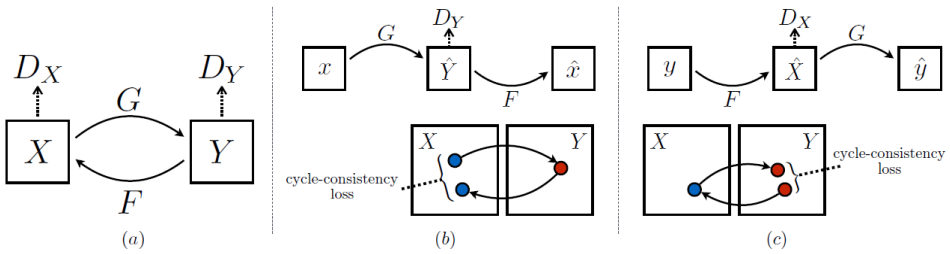


Figure 2.52: Architecture of cycleGAN (a) with cycle consistency loss showed in (b) and (c). [24]

Progressive growing

The key idea of *progressive growing* [25] is that the networks (discriminator and generator) are not trained with the full resolution. In progressive growing the networks are progressively growing, starting from a low resolution and adding new and new layers so that the model outputs more and more detailed results as training progresses. This technique is used in the approach proposed in this thesis.

In the early stages of the training, when training with low-resolution images, the networks learn the large-scale structure of the image distribution. As the training progresses, new layers are added to both discriminator and generator, higher resolution is used, and the networks learn finer and finer details. In an example of generating faces, this would mean that the network would learn how does a face look in general (positioning of eyes, nose, mouth,...) and later learn different hairstyles, eyelashes, etc.

Discriminator and generator are mirror images of each other and grow synchronously. The layers that are already in the networks remain trainable even after the addition of new layers.

The main benefits of progressive growing are:

- Generation of small images early in the network is more stable.
- Increasing the resolution step by step is an easier task then to generate the full resolution straight on.
- The training time is reduced.

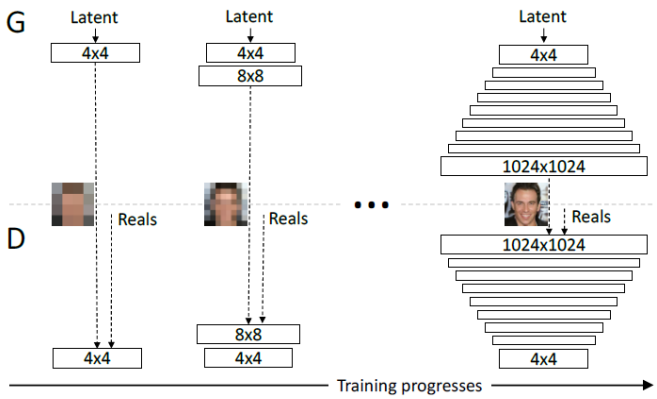


Figure 2.53: Both discriminator and generator grow synchronously starting with low resolution and proceeding by adding new and new layers all the way to the full resolution [25].

When a new layer is added and the resolution of both generator and discriminator doubles, this new layer is faded in smoothly. In the example in Figure 2.54 you can see a transition from 16×16 resolution in (a) to

32×32 resolution in (c). The transition phase is depicted in (b). During this translation the layers that operate on the higher resolution are treated like residual blocks, whose weight increases linearly from 0 to 1, meaning that in the early stages of the transition they have no or just a little effect. In Figure 2.54 2 and 0.5 refers to doubling and halving the image resolution using nearest neighbor filtering and average pooling, respectively. The *toRGB* block consists of 1×1 convolutions and is used for mapping the feature vectors to the RGB space. The *fromRGB* block does the reverse and uses the 1×1 convolutions as well.

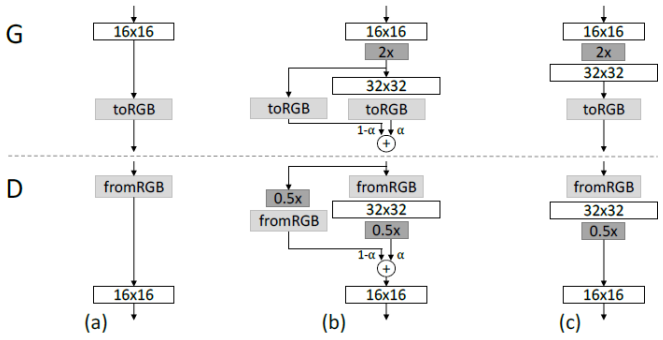


Figure 2.54: An example of a fade in of a new layer [25].

Other contributions by paper [25]:

- *Minibatch standard deviation layer*
  This is a parameter-free alternative to the Minibatch discrimination layer. The authors say that it "is implemented by adding a minibatch layer towards the end of the discriminator" [25].

- *Equalized learning rate.*
  The idea of equalized learning rate is to scale the weights dynamically scaled layer-wise by a constant from He's initializer [14]. This ensures that the dynamic range, and thus the learning speed, is the same for all weights [25].

- *Pixelwise Feature Vector Normalization in Generator*
  The goal of the pixelwise feature vector normalization is to prevent the scenario where the weights of generator and discriminator blow out. Thus, the authors propose to normalize the feature vector in each pixel to unit length *in the generator* after each convolutional layer.

2.5 Image Processing

In this section, I will briefly describe few commonly used techniques in image processing.

First, in section 2.5.1, I will describe the local binary patterns. Then, in the section 2.5.2, I will give a short introduction to how the histograms can be used in image processing along with examples of a few commonly used distance measures for comparison of image histograms.

Since the image processing is tightly connected to the notion of color, I give a brief introduction to a few standard color models in section 2.5.3

2.5.1 Local Binary Patterns

Local Binary Patterns [82] (LBP for short) is a type of image descriptor. It is defined as a gray-scale invariant texture measure. LBP is made invariant against the rotation of the image domain and supplemented with a rotation invariant measure of local contrast.

LBP is a simple yet powerful texture operator which labels the pixels of given image by thresholding the 3×3 neighborhood of each pixel with the value of the center pixel and considers the result as a binary number [26]. The resulting decimal number is given as

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(x)2^p, x = g_p - g_c \tag{2.78}$$

where g_c corresponds to the gray value of the center pixel at position (x_c, y_c), g_p is a gray value that corresponds to the one of P equally spaced pixels on a circle of radius R. Examples of different neighborhood sets can be seen in Figure 2.55. The operator s defines a thresholding function specified as

$$s(x) = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x < 0 \end{cases} \tag{2.79}$$

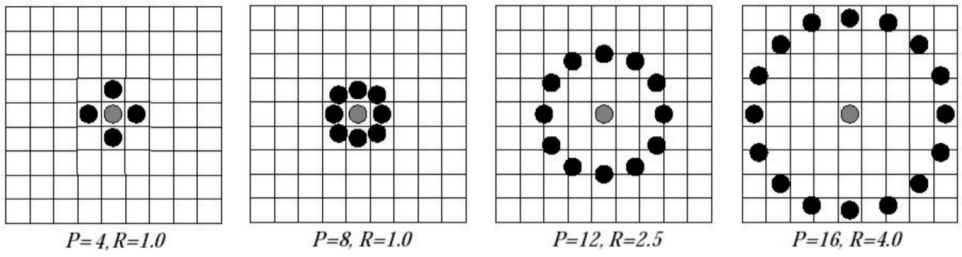


Figure 2.55: Neighborhood sets specified by different values of P and R. In the case that the sampling point is not in the center of a pixel, the pixel values are bilinearly interpolated [26].

An example of an LBP computation can be seen in Figure 2.56.

As it has been shown in [83], some bins contain more information than others and are called *uniform patterns* by authors. These patterns are specified by the fact that they contain 2 transitions $0 \rightarrow 1$ or $1 \rightarrow 0$. For example,

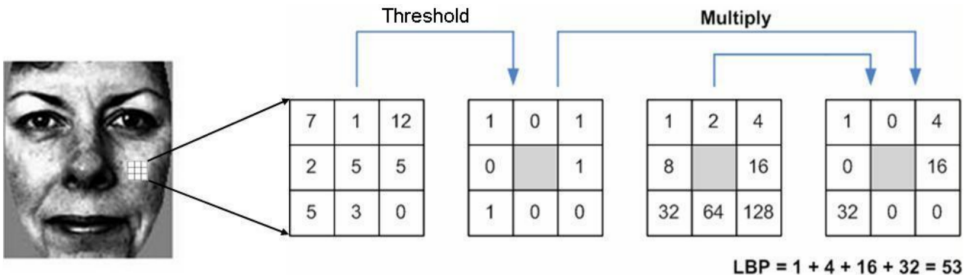


Figure 2.56: An example of LBP computation [26].

pattern 00000110 contains 2 such transitions and is therefore considered to be a uniform pattern, but 00100110 contains 4 transitions, which mean that it is not uniform. A standard LBP with 8 neighboring pixels has 256 different patterns and 59 patterns in "uniform" version.

2.5.2 Image histograms

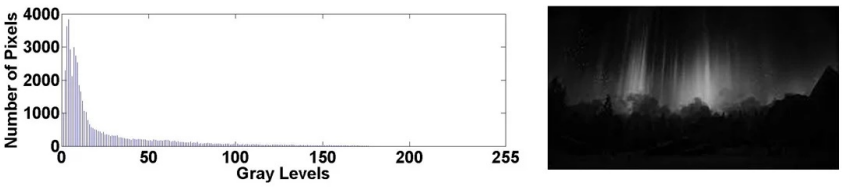
In a computer vision, it is common to compare the image histograms. An image histogram is a histogram of color intensities or of grayscale values. The x axis is divided into bins such that each bin corresponds to a certain range of intensity. The y axis then depicts the (relative) occurrence of intensities in a certain bin.

Based just on the histogram, we can say whether the image is bright or dark (figure 2.57), whether it has a high contrast (figure 2.58) or not or whether there are any (either dark or bright) saturated colors.

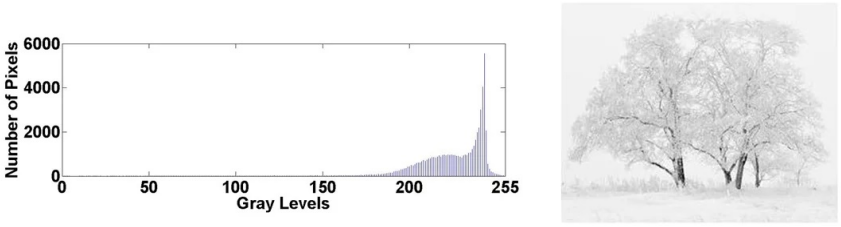
One of the main drawbacks of image histograms is that they encode no spatial information. Let us imagine two images. The first one has a checkerboard pattern of white and black squares. The other one has one half completely black and the other half completely white. These two images look very different, yet they would have the exact same image histogram.

Image histograms can be used e.g., for image enhancement. The goal of image enhancement is to make the image more visually appealing by using techniques as contrast stretching or histogram equalization.

However, we do not want to just look at a single image, but we would also like to use histograms for comparisons between different images. For this purpose, I will show a few commonly used distance measures for comparing the similarity of image histograms in the following sections.

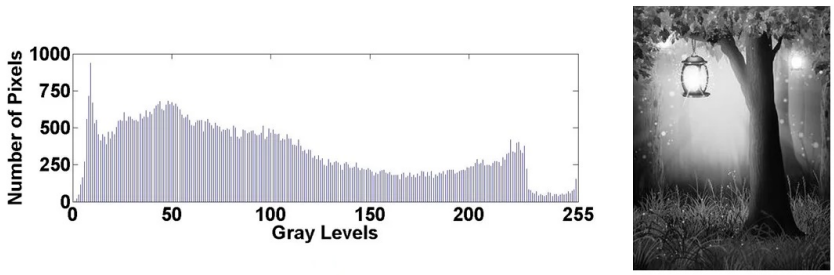


(a) : Histogram of a dark image.

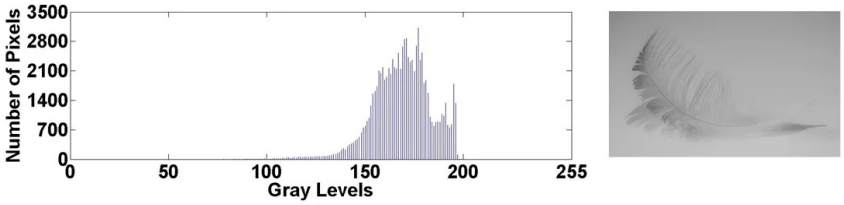


(b) : Histogram of a bright image.

Figure 2.57: Comparison of histograms of dark and bright image [27].



(a) : Histogram of an image with high contrast.



(b) : Histogram of an image with low contrast.

Figure 2.58: Comparison of histogram of image with high and low contrast [27].

Bhattacharyya distance

The *Bhattacharyya distance* is a measure between two probability distributions. This implies that the input histograms need to be normalized such that the values in the bins sum up to 1.

This distance is closely related to the *Bhattacharyya coefficient*, which

measures the overlap of two statistical samples and is used to measure the separability of classes in the problem of classification.

The Bhattacharyya distance is defined for two probability distributions p, q of the domain $\mathcal{X}$ as

$$D_B(p,q) = -\ln(BC(p,q)) \tag{2.80}$$

where the BC is the Bhattacharyya coefficient which is defined for discrete distributions as

$$BC(p,q) = \sum_{x \in \mathcal{X}} \sqrt{p(x)q(x)} \tag{2.81}$$

and for continuous distributions as

$$BC(p,q) = \int \sqrt{p(x)q(x)}\mathrm{d}x \tag{2.82}$$

In both cases it holds that $0 \leq BC \leq 1$ and $0 \leq D_B \leq \infty$. This distance does not obey the triangle inequality.

Hellinger distance

The *Hellinger distance* is other measure of similarity and is closely related to the Bhattacharyya distance since it can be defined using the *Bhattacharyya coefficient* (eq. 2.81 and 2.82) as

$$H(p,q) = \sqrt{1 - BC(p,q)} \tag{2.83}$$

where p and q are probability distributions.

The Hellinger distance obeys the triangle inequality (in contrast to the Bhattacharyya distance), and it holds that $0 \leq H(p,q) \leq 1$.

Earth mover's distance

The *Earth mover's distance* (EMD for short) is yet another measure of the distance between two probability distributions. It reflects the minimal amount of work that must be performed in order to transform one distribution into the other moving "distribution mass" around [84].

Given two distributions, one can be seen as a mass of earth spread adequately in space, the other as a collection of holes in that same space. We can always assume that there is at least as much earth as needed to fill all the holes to capacity by switching what we call earth and what we call holes if necessary. Then, the EMD measures the least amount of work needed to fill the holes with the earth. Here, a unit of work corresponds to transporting a unit of the earth by a unit of (ground) distance [84].

The computation of EMD is based on a solution to *transportation problem*. Transportation problem is can be defined as the linear program where $\mathcal{I}$ is

the set of suppliers, $\mathcal{J}$ is a set of customers and c_{ij} is the cost of shipping a unit supply from supplier $i \in \mathcal{I}$ to a customer $j \in \mathcal{J}$. The goal is to find such a flow f_{ij} that minimizes the overall cost

$$\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} c_{ij} f_{ij} \tag{2.84}$$

subjected to these constraints

$$\begin{aligned} f_{ij} &\geq 0, \quad && i \in \mathcal{I}, j \in \mathcal{J} \\ \sum_{i \in \mathcal{I}} f_{ij} &= y_j, \quad && j \in \mathcal{J} \\ \sum_{j \in \mathcal{J}} f_{ij} &\leq = x_i, \quad && i \in \mathcal{I} \end{aligned} \tag{2.85}$$

where where x_i is the total supply of supplier i and y_j is the total capacity of consumer j. Once the solution is found, the EMD distance is computed as

$$\mathrm{EMD}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} c_{ij} f_{ij}}{\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} f_{ij}} = \frac{\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} c_{ij} f_{ij}}{\sum_{j \in \mathcal{J}} y_j} \tag{2.86}$$

In the case of one-dimensional histograms, the EMD can be computed by dynamic programming as

$$\begin{aligned} \mathrm{EMD}_0 &= 0 \\ \mathrm{EMD}_{i+1} &= P_i + \mathrm{EMD}_i - Q_i \\ \text{Total Distance } &= \sum |\mathrm{EMD}_i| \end{aligned} \tag{2.87}$$

where P and Q are two histograms/distributions.

2.5.3 Color models

A color model is a mathematical model that describes the way of how colors can be represented as tuples of numbers. These tuples have most often three or four values.

When the model is associated with the interpretation of color components (conditions,...), the resulting set of colors is called *color space*. This space can be viewed as a region in n-D space, where n is the number of color components. In the case that the x, y, z axes are identified to the stimuli of long- (L), medium- (M) and short-wavelength (S) light receptors, then the origin, $(S, M, L) = (0, 0, 0)$ corresponds to the black color.

RGB

RGB color model is one of the best-known color models. This is mainly due to the fact that a lot of electrical devices used or are still using this model, such as televisions.

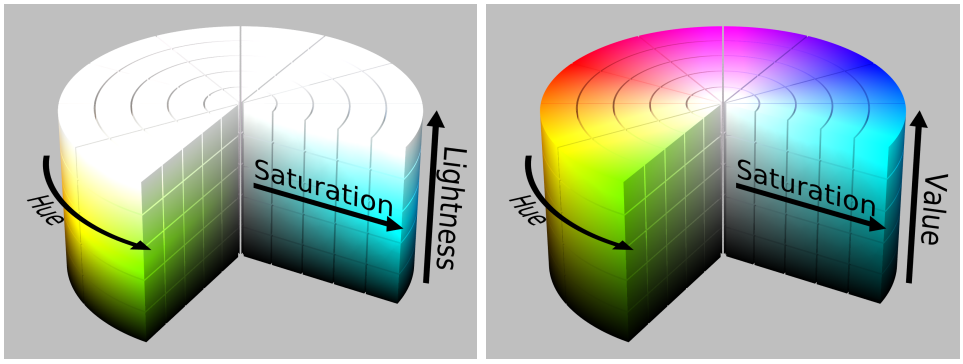
RGB is an additive model. This means that it uses additive color mixing with three primary colors: red (R), green (G) and blue (B). With a mixture of these three colors, one can cover a large part of the human color space.

HSV and HSL

HSV and HSL are two alternative representations to RGB model. The shortcut HSL comes from the tree components: *hue, saturation* and *lightness*. In the HSV model, the third component is different – it is called *value*. These two models were designed to better match the way of how people perceive color [28].

Both HSV and HSL have cylindrical geometries as can be seen in Figure 2.59. The hue component corresponds to the angular dimension. It starts with the red primary at 0°, continues through the green primary at 120° and through the blue primary at 240°.

In both HSV and HSL, the additive primary and secondary colors (red, yellow, green, cyan, blue and magenta) and linear mixtures between adjacent pairs of them are arranged around the outside edge of the cylinder with saturation 1 [28]. These colors have value 1 in HSV model and lightness 0.5 in the HSL model. You can see the HSV and HSL color cylinder in Figure 2.59.



(a) : HSL cylinder. **(b) :** HSV cylinder.

Figure 2.59: Comparison of HSL and HSV cylinders [28].

The four components of HSV and HSL are defined in [85] as follows:

- *Hue*: The "attribute of a visual sensation according to which an area appears to be similar to one of the perceived colors: red, yellow, green, and blue, or to a combination of two of them".
- *Saturation*: The "colorfulness of a stimulus relative to its own brightness".
- *Lightness / value*: The "brightness relative to the brightness of a similarly illuminated white".

Conversion from RGB. The conversion of H component is the same for HSV and HSL. However, the conversion from RGB to S component differs. This conversion to H is defined as:

$$H = \begin{cases} 0, & \text{if } MAX = MIN \Leftrightarrow R = G = B \\ 60° \cdot \left(0 + \frac{G-B}{MAX-MIN}\right), & \text{if } MAX = R \\ 60° \cdot \left(2 + \frac{B-R}{MAX-MIN}\right), & \text{if } MAX = G \\ 60° \cdot \left(4 + \frac{R-G}{MAX-MIN}\right), & \text{if } MAX = B \end{cases} \tag{2.88}$$

The conversion to S for HSV is defined as:

$$S_{\text{HSV}} = \begin{cases} 0, & \text{if } MAX = 0 \Leftrightarrow R = G = B = 0 \\ \frac{MAX-MIN}{MAX}, & \text{otherwise} \end{cases} \tag{2.89}$$

and for HSL as:

$$S_{\text{HSL}} := \begin{cases} 0, & \text{if } MAX = 0 \Leftrightarrow R = G = B = 0 \\ 0, & \text{if } MIN = 1 \Leftrightarrow R = G = B = 1 \\ \frac{\text{MAX}-MIN}{1-|MAX+MIN-1|} = \frac{2MAX-2L}{1-|2L-1|} = \frac{MAX-L}{\min(L,1-L)}, & \text{otherwise} \end{cases} \tag{2.90}$$

The value component V and lightness component L are defined as:

$$V = MAX \tag{2.91}$$

$$L = \frac{MAX + MIN}{2} \tag{2.92}$$

Chapter 3

State of the Art

In this section, I will briefly cover the state-of-the-art methods in relevant fields for this diploma thesis.

First, in the section 3.1, I will describe the most recent techniques used for image inpainting [77, 78, 86, 87, 88, 89, 81, 23, 90]. The goal of image inpainting is mainly in filling the holes in image such that the image would seem natural as a whole. This is relevant to the primary goal of this thesis. The only difference is in the fact that image inpainting tries to fill the gaps with the structures similar to the surroundings of the gap, while the goal of the thesis is to paint people into the given region and these people don't usually have the same texture as their surroundings. In this thesis, the information about the surroundings of the hole (where the person should be painted) can also be used, since it can, for example, bring information about the light conditions.

In the next section 3.2, I will describe a style-based architecture for GANs. The authors of the referred paper show the results of their approach on the example of face generation where they show the ability of the proposed solution to modify the output based on a given style. It uses the progressive growing of GANs (explained in section 2.4.3), which is one of the cornerstone techniques in this thesis. The proposed solution is a kind of conditional generation that is closely related to this thesis just as the section 3.3, where I describe a solution to image generation conditioned on input label maps that specify the classes of painted objects. These ideas are further improved by the network called *GauGAN* that is described in section 3.4 and which proposes a novel normalization layer that has the conditioned label map as an input.

In the last two sections 3.5 and 3.6, I present two approaches in generating person images. The paper presented in section 3.5 deals with pose guided image generation. This approach is used to generate a person in a given image in a novel pose. In the following section 3.6, the authors propose a framework that is able to manipulate the foreground, background, and person pose in the image.

3.1 Image inpainting

One of the most recent works in the field of image inpainting comes from the paper called *Free-Form Image Inpainting with Gated Convolution* [29], where the authors further improve the architecture of [23] and propose a solution to free-form inpainting, i.e. to filling the holes of non-rectangular shape in the image. Their solution is based on *gated convolutions* that, in contrast to vanilla convolutions, provide a learnable feature selection mechanism. The motivation behind introducing gated convolution is that in the case of vanilla convolutions all the pixels (disregarding whether they should be generated or not) are treated the same. This is not well suited for the task of image inpainting since the input features are composed of both regions with valid pixels outside holes and invalid pixels (shallow layers) or synthesized pixels (deep layers) in masked regions [29]. The proposed gated convolution learns a soft mask that is applied to the result of convolutional operation. This can be expressed as:

$$\begin{aligned} Gating_{y,x} &= \sum\sum W_g \cdot I \\ Feature_{y,x} &= \sum\sum W_f \cdot I \\ O_{y,x} &= \phi\left(Feature_{y,x}\right) \odot \sigma\left(Gating_{y,x}\right) \end{aligned} \tag{3.1}$$

where σ is a sigmoid activation making sure that the gating values are in range $[0, 1]$, ϕ can be any activation, $Gating$ are gating values applied as a soft filters to $Features$ and W_g and W_f are two different convolutional filters. The procedure can be seen in 3.1.

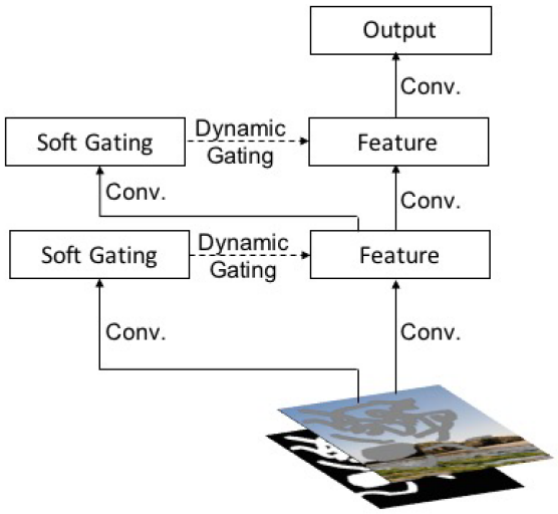


Figure 3.1: Illustration of gated convolution [29].

In addition to the input channel that denotes the mask, one additional channel corresponding to the sketch that the network should condition on

can be used. The overall architecture is shown in 3.2 along with visualization and interpretation of learned gating values. You can see that the network learns to attend more to sketch or mask in different depths of network and even to perform foreground/background semantic segmentation.

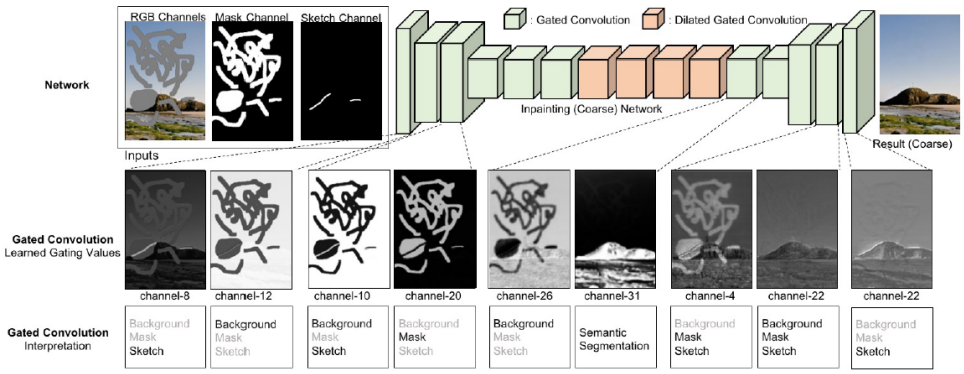


Figure 3.2: The architecture of [29] together with an example of learned gating values. Note that is figure show only the coarse part of the inpainting framework and not the refinement part.

This work provides an extension to User-Guided Image Inpainting by the use of sketches. The network is trained to be guided by these sketches during inpainting of missing regions. The authors showed this extension on the example of natural images which can be seen in 3.3. To obtain training data for this task, they use the HED edge detector [91] and then set all values that are above some predefined threshold to ones. These binary edge images then serve as training sketches.

3.2 StyleGAN

In the paper called "A Style-Based Generator Architecture for Generative Adversarial Networks" [30], the authors further improve the idea of progressive growing of Generative Artificial Networks used for face generation by proposing a generator architecture that is able to learn an unsupervised separation of high-level attributes (such as pose and identity) and stochastic variation (e.g., changes in hairstyle, position of freckles,...). The proposed architecture enables to control these attributes intuitively.

In the previous works, the input to the first layer of the generator is a latent code. In this work, they depart from this approach and use a learned constant input straight on. The *style* of the figure is encoded by a non-linear mapping network $f : \mathcal{Z} \to \mathcal{W}$ that maps the latent vector z coming from the latent space $\mathcal{Z}$ to a vector $w \in \mathcal{W}$, where $\mathcal{W}$ is another latent space. The vector w is then transformed by learned affine transformations to *styles* encoded as $y = (y_s, y_b)$ that control adaptive instance normalization (AdaIN for short) operations after each convolution layer of the synthesis network g

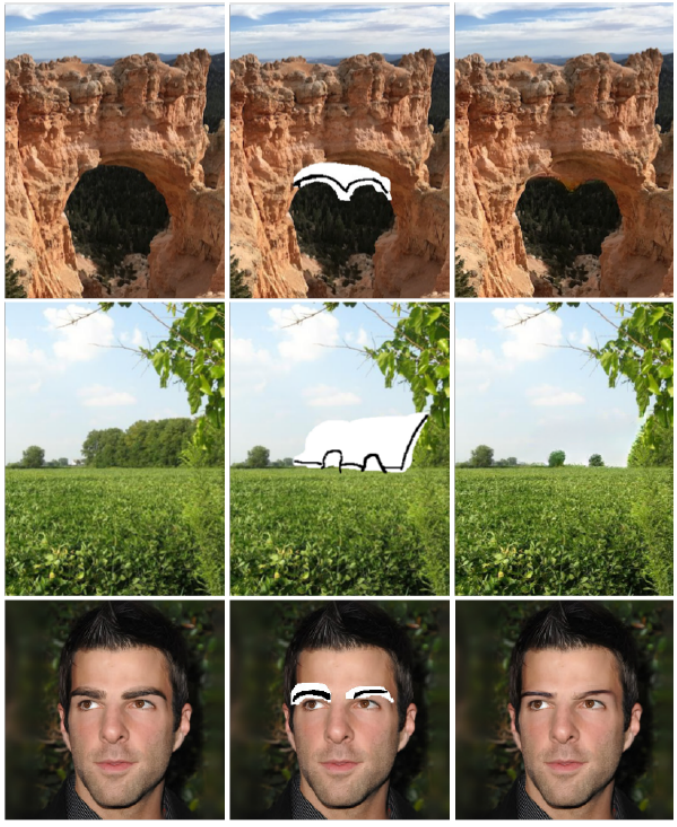


Figure 3.3: An example of user-guided image inpainting by sketches [29].

[30]. The AdaIN operation is defined as

$$\text{AdaIN}\left(\mathbf{x}_i, \mathbf{y}\right) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu\left(\mathbf{x}_i\right)}{\sigma\left(\mathbf{x}_i\right)} + \mathbf{y}_{b,i} \tag{3.2}$$

where each feature map x_i is first normalized, then scaled by y_s and biased with y_b.

The mapping network f together with the affine transformations A and AdaIN operation controls the *styles* of the generated image. The stochasticity is injected to the image from single-channel noise images. This noise input is broadcasted to all feature maps by learned per-feature scaling factors and added to the output of the corresponding convolution [30]. The architecture is shown in Figure 3.4 and a few examples can be seen in Figure 3.5.

3.3 pix2pixHD

In the paper "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs" [31], the authors present a new image-to-image method for synthesizing high-resolution images from semantic label maps.

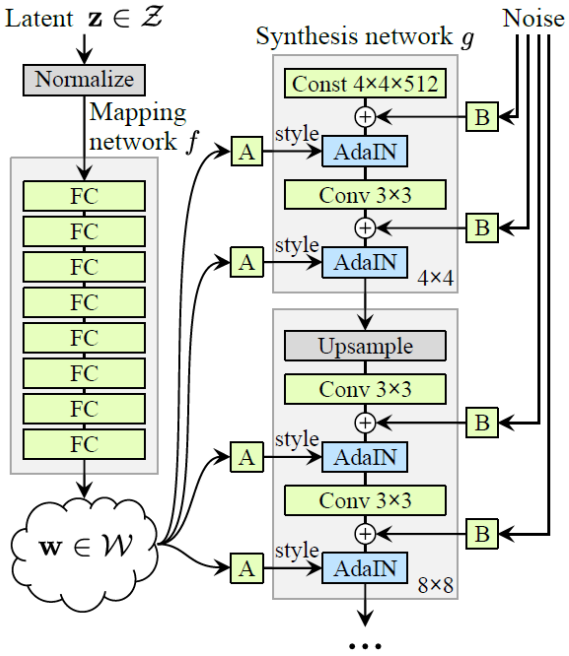


Figure 3.4: Architecture of style-based generator [30].

Figure 3.5: Examples of generated faces [30].

To achieve this, the use of coarse-to-fine generator that is decomposed to

two generators. These two generators finally produce 2048×1024 image from a given label map. This structure is shown in Figure 3.6.

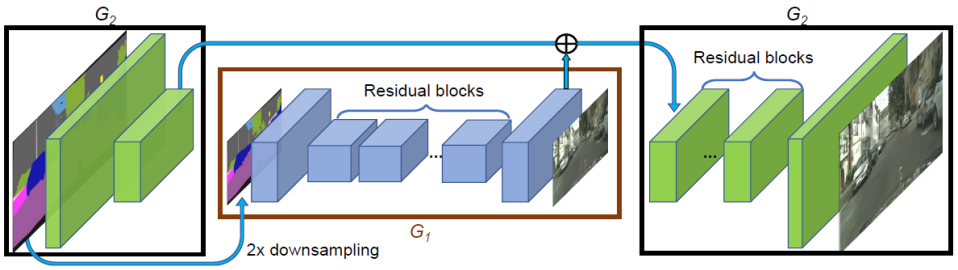
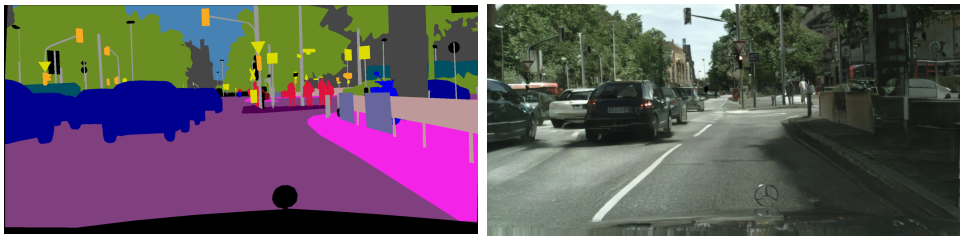


Figure 3.6: Architecture of the generator in pix2pixHD [31].

They use a multi-scale version of discriminator that operates at different levels of detail. This multi-scale discriminator consists of 3 identical discriminators. The first discriminator works with the original input (the finest details) and encourages the generator to produce finer details. The two other discriminators work with coarser images that are downsampled by the factor of 2 and factor of 4 respectively.

They also add a feature-matching loss (2.4.1) on features obtained from the discriminator – this loss is optimized by the generator only, and the discriminator serves just as a feature extractor.

Another feature that helps the model to generate better results is an addition of the boundary map to the input label map. This enforces clear boundaries between instances.



(a) : label map **(b) :** generated image

Figure 3.7: An example of input label map in 3.7a and resulting generated image in 3.7b [31].

3.4 GauGAN

The main novelty of paper called "Semantic Image Synthesis with Spatially-Adaptive Normalization" [32] is in the proposed *spatially-adaptive normalization* (*SPADE* in short), which is a normalization layer that should help to produce better images given an input semantic layout. They claim that most of the previous methods that directly feed the semantic layout as input to

the deep network and process this input with a stack of convolution, normalization, and nonlinearity layers are at most suboptimal as the normalization layers used in such networks tend to wash away the semantic information. Note that this can be different from the approach used in this thesis since the "semantic information" used in this work consists only of values in range $[0, 1]$ and should only be used for guidance of which pixels should be changed.

Proposed normalization layer SPADE applies spatially-varying affine transformation and is therefore well-suited for the task of image synthesis from semantic mask. In this layer, the input activations are first normamalized in channel-wise manner and then modulated with learned scale and bias [32]. The design of spade in shown in Figure 3.8. Input semantic segmentation mask is $m \in \mathbb{L}^{H \times W}$, where $\mathbb{L}$ is a set of integers denoting the semantic labels, and W and H are image width and height. This layer can be mathematically described as follows:

$$\gamma^i_{c,y,x}(m)\frac{h^i_{n,c,y,x} - \mu^i_c}{\sigma^i_c} + \beta^i_{c,y,x}(m) \tag{3.3}$$

where $h^i_{n,c,y,x}$ is the input activation and μ^i_c and σ^i_c are the mean and standard deviation of the activation in channel c.

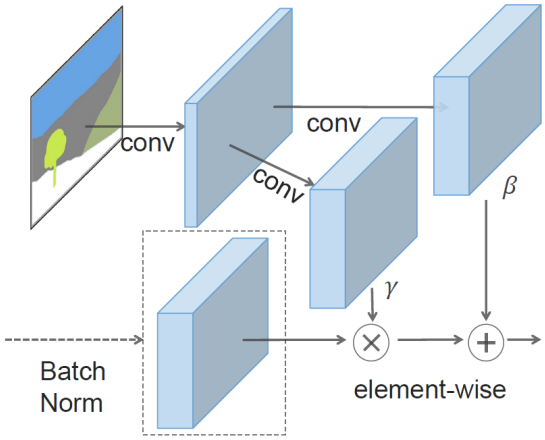


Figure 3.8: Design of SPADE layer [32].

The important thing to note in equation 3.3 is that the variables $\gamma(m)$ and $\beta(m)$ of the normalization are learned and depend on the input semantic mask and position. These variables are, in fact, functions that convert the input semantic mask m into scaling and bias values and are implemented as a simple two-layer convolutional networks.

The difference of generator trained with SPADE compared to the classic conditional generator is in fact that it is no longer needed to feed the segmentation map to the first layer. Therefore, the encoder part of the generator is discarded since the learned modulations in SPADE contain enough information about the layout. The generator can take a random noise vector as

an input, which results in stochasticity of outputs. The architecture of a generator with SPADE is shown in Figure 3.9

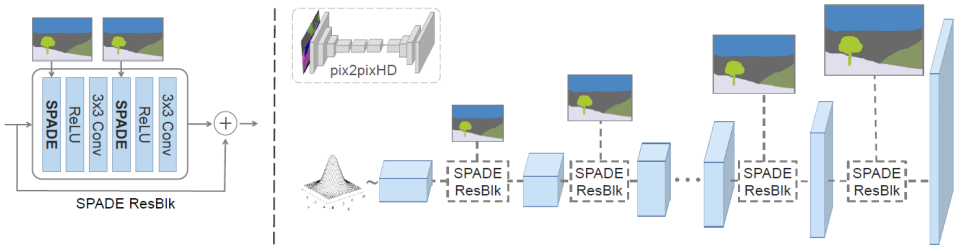


Figure 3.9: On the left, you can see a structure of a SPADE residual block. (Right) Design of SPADE generator consisting of residual blocks [32]. In comparison to image-to-image translation networks (such as pix2pixHD), this network has less parameters and better performance.

3.5 Pose Guided Person Image Generation

In the paper called "Pose Guided Person Image Generation" [33], the authors propose a novel network called *Pose Guided Person Generation Network* that synthesize person images in arbitrary poses, based on the image of that person and a novel pose. This network consists of two stages: pose integration and image refinement.

In the first stage, they use a variant of U-Net (2.2.10) to integrate the target pose with the person image. There is a fully-connected layer in the bottleneck of this network as you can see in Figure 3.10. The input to this network consists of a conditioned image and a pose encoded by 18 heatmaps (one heatmap per joint/keypoint). In each of those heatmaps, the keypoint is represented as a radius of 4 pixels around the keypoint position. The input is then a concatenation of conditioned image and these heatmaps. The output is a coarse image that captures the global structure of the human pose in the given target image. The change of the background between conditioned and target image is enforced through L1 loss. Using the L1 loss results in blurry images since it encourages the outputs to be an average of all possible cases [73].

The output of the first stage serves as an input to the second stage. In that stage, they use a variant of DCGAN (2.4.1) to improve the results further. Here, the refining generator takes as an input the image conditioned on and the coarse image generated in the first stage. The goal of this generator is to output an *appearance difference map* that would bring the coarse and blurry result closer to the conditioned image. There is no fully-connected layer in this generator (opposed to the generator in stage one) which helps to preserve more details. The use of difference maps is claimed to speed up the convergence since the network only focuses on learning the missing details.

The architecture of both generators is shown in Figure 3.10.

The discriminator is trained to recognize between real and fake pairs. The real pair consists of condition image and the target image. The fake pair contains the output of the second generator instead of the target image. This pairwise input to the discriminator encourages it to learn the distinction between the generated and target image instead of just to distinguish between real and synthesized images. This process can be seen in Figure 3.10.

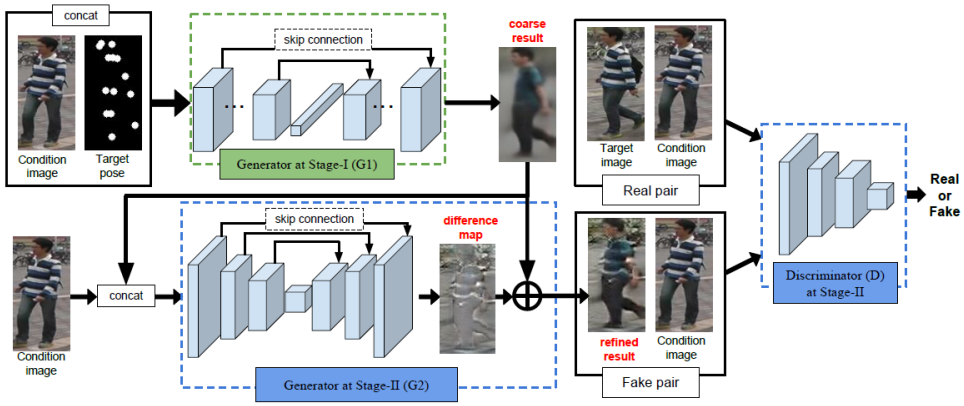


Figure 3.10: The architecture of Pose Guided Person Generation Network from [33].

3.6 Disentangled Person Image Generation

The paper "Disentangled Person Image Generation" [34], the authors aim at generating novel, yet realistic, images of persons. To achieve this, they use a two-stage reconstruction pipeline that learns a disentangled representation of image factors and generates novel person images.

In the first stage, they propose a multi-branch reconstruction architecture – there is a branch for foreground, background, and pose. Using this architecture, the goal is to disentangle the foreground, background, and pose factors from each other. In the *foreground branch*, they apply the corse pose mask to the feature maps first, then arrange the body part feature embeddings according to seven region-of-interest (ROI for short) bounding boxes obtained from pose keypoints. These ROIs are of size 48×48 and are fed to encoders with shared weights. This results in a set of 32D feature vectors that are concatenated to one 224D feature vector for the foreground. In the *background branch*, they inverse the pose mask and obtain one 128D feature vector that is then concatenated and tiled with a foreground feature vector to one $128 \times 64 \times 352$ appearance feature map. In the *pose branch*, they concatenate the 18-channel heatmaps that correspond to the locations of keypoints with the appearance feature maps. This input is passed into the "U-Net"-based architecture, i.e., convolutional autoencoder with skip

connections, to generate the final person image following [33] (sec. 3.5).

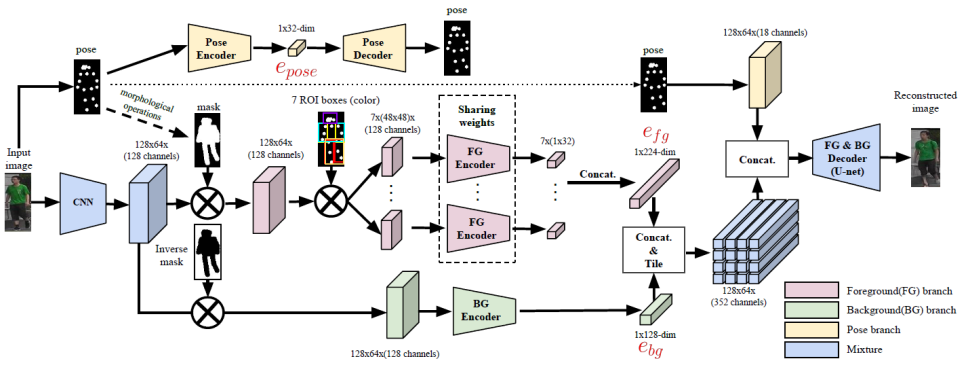


Figure 3.11: Architecture of the first stage [34].

It is known that images can be represented by a low-dimensional, continuous feature embedding space since they lie on or near a low-dimensional manifold of the original high-dimensional space [33] [92, 93]. Based on this observation, the authors suppose that the distribution of such feature embedding space should be more continuous and easier to learn and propose a two-step mapping technique that is illustrated in Figure3.12. In the first step, they learn a mapping function Φ that maps from Gaussian space $\mathcal{Z}$ into a continuous embedding space E and then use the pretrained decoder from the first stage to map from the embedding space $\mathcal{E}$ to the space of real images $\mathcal{X}$ [33]. As was written earlier and as can be seen in Figure 3.11, the encoder in the first stage encodes the foreground, background and pose to lower dimensional vector embeddings e. The mapping function Φ then learns to map from Gaussian noise $z \in \mathcal{Z}$ to the space of embeddings E in an adversarial manner. Using this, we can sample *fake* embedding features from Gaussian noise and map them to the space of real images by the encoder learned in the first stage.

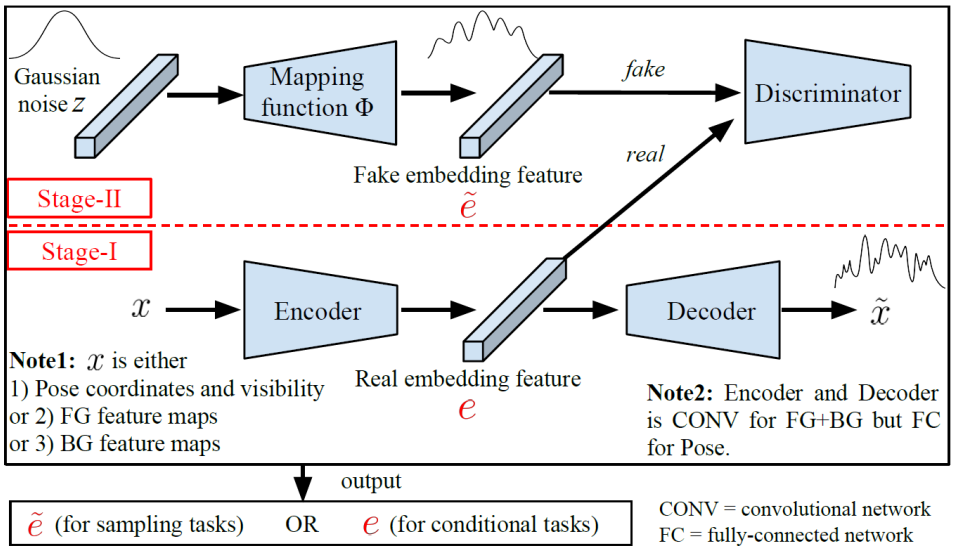


Figure 3.12: Architecture of the whole framework of [34].

Part II

Implementation and experiments

Chapter 4

Datasets

For the purpose of this thesis, two datasets were created. One is used for classification (section 4.2) and is based on the Cityscapes dataset [94] and the other one based on the MS COCO [35] (described in the section 4.2) dataset is intended for the training of person image generation.

The reason behind the creation of the dataset for person image generation is mainly the lack of suitable datasets that can be used for such a task. The current datasets available are either small in size, small in resolution, or do not provide important information such as person segmentation and keypoint locations.

In the first section of this chapter, the MS COCO dataset is briefly introduced. In the next sections, I describe the process of the two datasets.

4.1 MS COCO dataset

The Common Objects in Context (COCO) is a large-scale dataset that tackles the problems in scene understanding from detection to localization [35]. In contrast to other datasets, this one contains objects in various poses, from multiple views, in different backgrounds and even partially occluded.

The dataset was created with extensive use of Amazon Mechanical Turk. First, the authors needed to gather a broad set of images that would contain contextual relationships and non-iconic object views. Then, each of the images was labeled as containing particular object categories, and for each such category found, the individual instances of the category in the image were labeled and segmented [35].

The dataset contains 91 common object categories and out of those 82 have more than 5000 labeled instances. In total, the dataset contains 2.5 million instances in 328 thousand images.

The categories labeled in the dataset are only "thing" categories and not

"stuff". A "thing" categories contain objects whose instances can be easily labeled (humans, bicycles, dogs,...) where "stuff" categories contain objects with no clear boundaries (sky, road,...).

One of the most significant advantages of this dataset is that it does not contain object instances in the iconic situations only, i.e., image with a single dominant object in the center of the image. It contains the instances in non-iconic situations and therefore helps the algorithms trained on it to generalize better. Examples of annotated images can be seen in Figure 4.1. It also contains keypoints of person pose, which comes handy while creating the dataset described in the following section.

Figure 4.1: An example of annotated images in MS COCO dataset [35].

4.2 Created datasets

As it was mentioned in the introduction, for the purposes of this thesis, two datasets were created. The first dataset was created for the task of classification, and the second dataset was created for the task of person generation.

Dataset for classification

This dataset is intended for the evaluation of an impact of various data augmentation methods on image classification and is created for the task of determining whether the given image contains a person or not.

The dataset is based on the Cityscapes dataset [94] that provides information about instance segmentation of the scene. From these segmentations, pedestrian positions are determined and cropped. The pedestrian must be at least $height_{min}$ high (can vary, 100px in the experiments) and the resulting crops are resized to 224×224px. By this process, a total of n samples is obtained. However, all these samples belong to the positive class, i.e., contain

a pedestrian. In order to have a balanced set, n random crops that do not contain pedestrian are created and added to the dataset.

The dataset is split to the tree traditional splits used for training, validation, and testing. The size of the dataset splits for $height_{min} = 100$ is show in table 4.1.

split	*number of samples*
train	10976
val	1100
test	1100

Table 4.1: Size of the dataset for classification with $height_{min}$ set to 100px.

Exemplary samples for both positive and negative class are shown in Figure

Figure 4.2: An example of possitive (top) and negative (bottom) samples from the dataset for the detection task.

Dataset for person generation

As it was already mentioned earlier, the dataset created for the purpose of person image generation is based on the MS COCO dataset described in the previous section 4.1. This section describes the process of creation.

First, only the images with *person* label are chosen. This is quite simple with the use of API that is provided by the authors of the dataset.

Each annotated person in the image must suffice multiple criteria in order to be included in the dataset. First, it is checked whether it is at least as high as the predefined minimal threshold. If not, such an annotation is discarded and not taken into account later. It is important to note that the bigger threshold will allow learning to generate images in higher resolution, but

the resulting dataset will contain fewer samples. Another criteria that the annotated person must suffice is that it must be standing. This is determined just based on the ratio of the *height* and *width* that should be at least as high as the predefined ratio. This ratio was empirically set to 2. Another important thing is that it is required that at least one keypoint on the head, shoulder, elbow, hip, and knee is visible.

The dataset contains cropped images from the original ones with person annotations. An example of the cropped images can be seen in Figure 4.3. From these examples, it can be clearly seen that the variety of poses, situations, lighting conditions, occlusions, and human clothing is enormous. This makes the dataset very challenging.

Figure 4.3: An example of the cropped images from the dataset.

Each such samples comes along with other usable content:

1. *Masked image*: it is the same cropped image just with the person masked out.
2. *Mask* is a binary (or grayscale) one-channel image corresponding to the segmentation mask of the masked-out person.
3. *Estimated mask* is a mask estimated from the keypoints. The estimated mask is not just binary with $\{0, 1\}$ values. This is because the estimation from the keypoints cannot capture the person segmentation perfectly and therefore, a kind of a *soft* mask with values in the range $[0, 1]$ is used.
4. *Keypoint locations* are given as a 17-channel tensor with each channel corresponding to the location of a single keypoint in the image.

An example of the entire batch from the dataset is shown in Figure 4.4.

The dataset is further enlarged by random horizontal flips and random grayscale conversion. The split to training and validation split follows the original split of MS COCO dataset n the version from the year 2017.

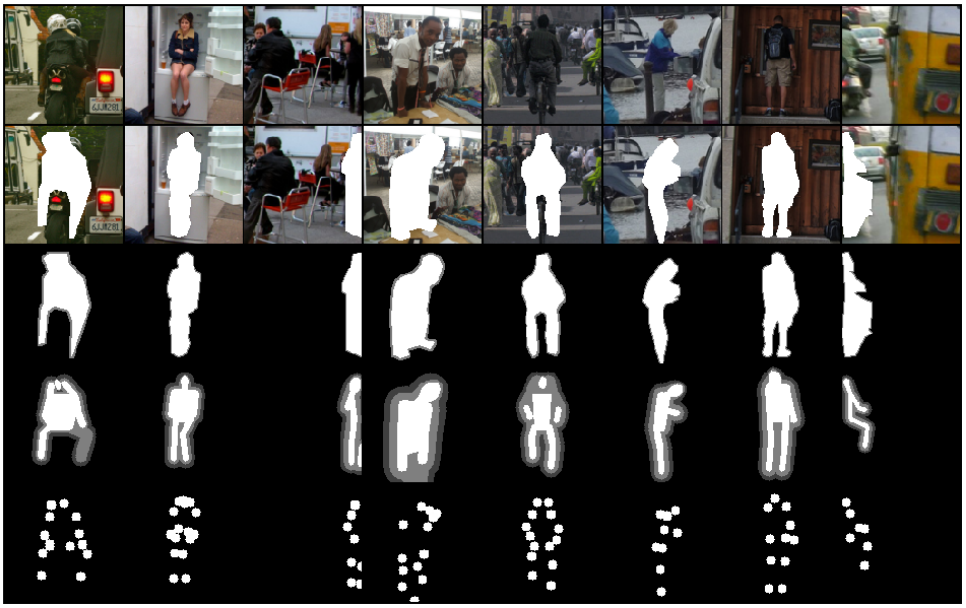


Figure 4.4: An example from the dataset. The top row shows the original cropped image, the second row shows a masked version of the image, the third image shows the original mask, while on the fourth the estimated mask is shown and in the last row are shown the positions of annotated keypoints.

Mask estimtion. The process of mask estimations is based only on the information about keypoints position and visibility. The first step is drawing the connections between neighboring keypoints in the skeleton. A connection is drawn if and only if at least one of the neighboring keypoints is annotated. The process of mask creation is described in algorithm 1. The results can be seen in the fourth row in Figure 4.4.

Cityscapes dataset. The dataset for person generation can even contain data form the Cityscapes dataset. This dataset contains instance-level segmentation that was used to obtain ground-truth person masks. However, it does not contain any information about keypoint locations. To deal with this missing part, I use the OpenPose [95] detector for keypoint estimation. It needs to be noted that this detector doesn't have to produce very accurate results. This is one of the reasons why this part of the created dataset was not used during the experiments.

Result: Estimated mask $\mathcal{M}$
input: skeleton edges E, keypoint locations $\mathbf{x}$, keypoint visibility $\mathbf{v}$;
initialize $\mathcal{M}$ with zeros;
initialize the widest, medium and the narrowest line width $\mathbf{w}$ with line values determined by function $f(width) : \mathbf{w} \rightarrow [0, 1]$;
for *line width* w *in* [*widest, medium, narrowest*] **do**
  for $(i, j) \in E$ **do**
    if *both keypoints* i *and* j *are annotated* **then**
      if *not both keypoints are visible and* w==*"narrowest"* **then**
        continue;
      end
      draw line from x_i to x_j of width w and value $f(w)$ to mask $\mathcal{M}$;
    end
  end
end

Algorithm 1: An algorithm for drawing a mask based only keypoint locations only.

Chapter 5

Experiments with augmentation techniques

In this section, I will describe the experiments with data augmentation technique. These experiments will be evaluated on a patch classifier which can be run on top of the pedestrian detection pipeline. This is a common approach in autonomous driving when CNN-based methods are used only in the last stage of a pipeline when there is only a small amount of samples since the hardware used in the cars is often too slow.

5.1 Pedestrian detectors in cars

The pedestrian detection (PD) systems used in the cars have to be really fast and have only small computational requirements. This implies that the CNN-based methods cannot be used on the whole image since they would require too much computational power. Instead, the CNN model is used at the end of the PD pipeline. This is done mostly because the CNN inference generally takes more time than, e.g., boosting methods used in previous parts of the pipeline.

PD pipeline used in cars is often a cascade classifier [96] that sequentially reject false positive samples as it can be seen in Figure 5.1.

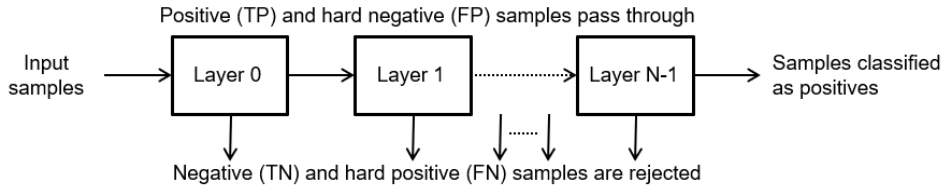


Figure 5.1: An example of a cascade classifier.

Each layer of it can be configured independently and can use a different classifier. It can look for example as follows (Figure 5.2):

1. *AdaBoost cascade*: A cascade of AdaBoost [97] detectors that sequentially

reject samples. Such a cascade of AdaBoost classifiers is very fast.

2. *Dense HOG reginement*: By using stronger features such as Histogram of Oriented Gradients (HOG) [98], it is possible to make the localization accuracy more robust.

3. *Tracking*: A tracking of pedestrians can be used. This results in more accurate and robust localization and can be used in a loop with a final CNN-based classifier.

4. *CNN classifier*: A CNN is run at the end of the pipeline only over a small number of image patches that were not rejected in the previous stages of the cascade.

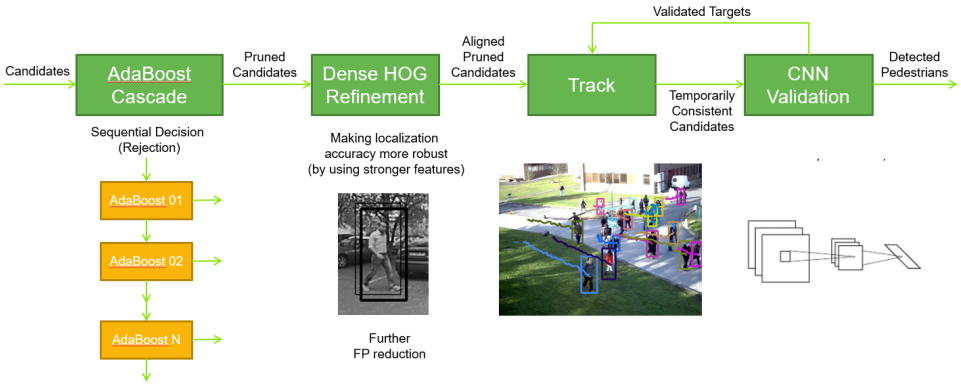


Figure 5.2: An example of pedestrian detection pipeline for candidate generation and sequential elimination of false positives.

The goal of the following experiments is to evaluate the contribution of different augmentation techniques on a CNN classifier of image patches that either do or don't contain a pedestrian. Such a classifier can be used in a PD pipeline, as mentioned above.

5.2 Implemented augmentation techniques

Most of the implemented and tested augmentation techniques were already described in section 2.3.

The used techniques can be divided into the ones that are applied to the whole image and to the ones that are applied only to pedestrian instances. These augmentations are applied to the whole image are:

- *Translation*: Translation of given patch in the image by a random value within a specified range.
- *Rotation*: Rotation of an image patch by a random angle drawn randomly from a predefined interval.

- *Random resized crop*: This technique first crops a random image (of size from predefined range) from the original patch, then randomly changes the aspect ratio of the cropped image and finally resizes it to a size of the original patch.
- *Flip*: Horizontal flip is applied with a probability $p \in [0, 1]$.
- *Random noise addition*: A random value drawn from a Normal distribution with zero mean is added to each pixel.
- *Color jitter*: Randomly change the brightness, contrast, saturation, and hue of an image.

On top of these augmentations that are standardly applied to the whole image, I experimented with augmentations to the pedestrian instances only. This requires to have an instance segmentation of the pedestrians in the image. If such segmentation is available, the image transformation is applied only to the pixels that belong to the chosen instance. I experimented with random noise addition and changes of brightness, contrast, saturation and hue.

5.2.1 Setup of experiments

Each row of the table 5.1 contains a name of the experiment and augmentations that were performed in this setup. The range specified with some of the methods stands for the parameters of the uniform distribution from which the random values are drawn. The shortcuts in the settings stand for:

- t [px]: maximal translation in pixels
- r [deg]: maximal rotation in degrees
- *scale*: scaling range as a ratio of the new size to the original size
- *aspect*: the range of change of the aspect ratio of the image
- *flip*: the probability of a horizontal flip
- *br*: the fraction of maximal brightness change
- *contr*: the fraction of maximal contrast change
- *sat*: the fraction of maximal saturation change
- *hue*: fraction of maximal hue change
- $\mathcal{N}(\mu, \sigma)$: parameters of the normal distribution used for noise addition
- *instance only*: denotes whether the augmentation was applied to the whole image or to the pedestrian instance only.

experiment name	*setting*										
	t [px]	r [deg]	scale	aspect	flip	br	contr	sat	hue	$\mathcal{N}(\mu, \sigma)$	instance only
baseline	0	0	(1.0, 1.0)	(1.0, 1.0)	0	0	0	0	0	(0,0)	×
translation	50	0	(1.0, 1.0)	(1.0, 1.0)	0	0	0	0	0	(0,0)	×
rotation	0	10	(1.0, 1.0)	(1.0, 1.0)	0	0	0	0	0	(0,0)	×
random resized crop	0	0	(0.8, 1.2)	(0.9,1.0)	0	0	0	0	0	(0,0)	×
flip	0	0	(1.0, 1.0)	(1.0, 1.0)	0.5	0	0	0	0	(0,0)	×
jitter, whole	0	0	(1.0, 1.0)	(1.0, 1.0)	0	0.1	0.1	0.1	0.1	(0,0)	×
jitter, person	0	0	(1.0, 1.0)	(1.0, 1.0)	0	0.1	0.1	0.1	0.1	(0,0)	✓
noise, whole	0	0	(1.0, 1.0)	(1.0, 1.0)	0	0	0	0	0	(0,3)	×
noise, person	0	0	(1.0, 1.0)	(1.0, 1.0)	0	0	0	0	0	(0,3)	✓
complete	50	10	(0.8, 1.2)	(0.9, 1.0)	0	0.5	0.1	0.1	0.1	(0,3)	both

Table 5.1: Table containing names of performed experiments together with their setup.

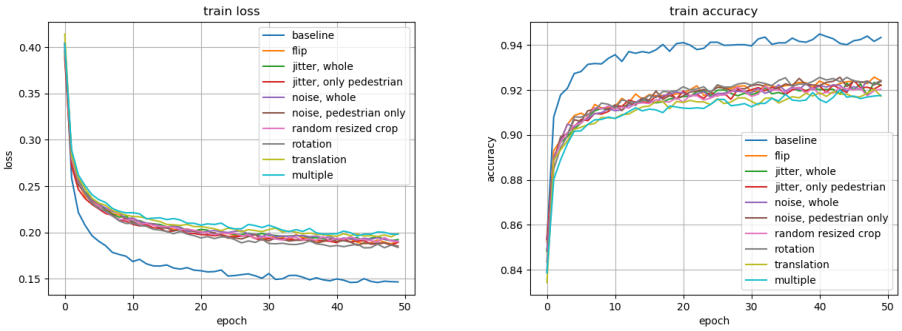
5.3 Comparison of augmentation methods

In this section, I will compare the performance of a CNN classifier when trained in the same setup just with different augmentation methods applied to the training dataset. The goal of this classifier is to classify whether the given patch contains a pedestrian or not. For this, I use the created dataset explained in 4.2.

As a CNN, I choose a VGG-like network pretrained on a classification task with ImageNet dataset. Note that this dataset does not have a label for humans or for pedestrians. I use a transfer learning to retrain this network to the specified task. To be precise, I remove the final classification layer of the original network that classifies based on a 4096-D feature vector and replace it with a new layer that takes the same number of features and outputs just one value. This one value is then passed to the sigmoid function and represents confidence of a network that the given image contains a pedestrian.

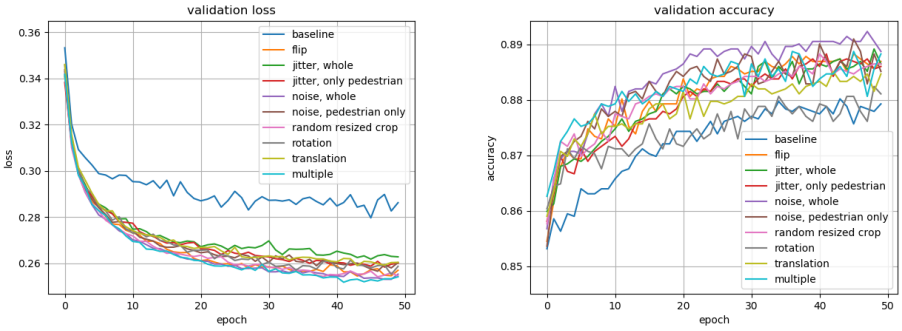
For each setup, I train this network with binary cross entropy loss for 50 epochs with Adam optimizer with learning rate 0.0001 and with a batch size 64. During the training, I save the network weights in the case that the loss/accuracy on the validation set decreases. After the training is complete, I load the network with weights that performed the best on the validation set and test its performance on the test set.

The learning curves for the training are shown in Figures 5.3and for validation in Figures 5.4. The performance on the test set is shown as a bar plot in Figure 5.5 with numerical values in table 5.2.



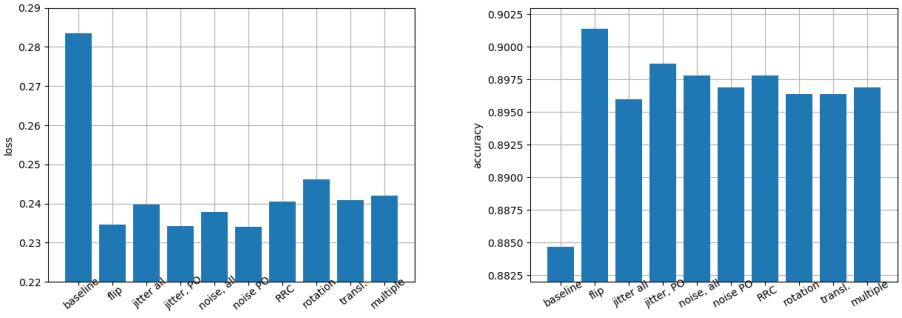
(a) : Loss. **(b) :** Accuracy.

Figure 5.3: Loss and accuracy on the training set.



(a) : Loss. **(b) :** Accuracy.

Figure 5.4: Loss and accuracy on the validation set.



(a) : Loss. **(b) :** Accuracy.

Figure 5.5: Loss and accuracy on the test set. PO stands for Pedestrian Only and RRC for Random Resized Crop.

From these results, it is clear that the baseline method with no augmentation applied to the training set is already pretty strong. This is due to the use of transfer learning with a network pretrained on a large ImageNet dataset. This network provides strong features that can be well distinguished by the classification layer. Another reason is that the dataset is not very challenging

experiment name	*loss*	*accuracy*
baseline	0.2834	0.8847
translation	0.2408	0.8964
rotation	0.2461	0.8964
random resized crop	0.2404	0.8978
flip	0.2346	**0.9014**
jitter, whole	0.2398	0.8960
jitter, pedestrian only	0.2343	0.8987
noise, whole	0.2378	0.8978
noise, pedestrian only	**0.2340**	0.8969
multiple	0.2420	0.8969

Table 5.2: Table showing the loss and accuracy of individual experiments on the test set.

by itself. The lighting conditions do not change too much, and the variance of pedestrian poses and clothing is not very huge, and the benefit of the network from augmenting this dataset is therefore not that large.

Despite this, we can see improvements to the baseline method in the experiments using data augmentation to the training data. All the experiments using some kind of augmentation methods achieve better results on both validation (Figure 5.4) and test set (Figure 5.5). The best accuracy is achieved with just simple horizontal flipping and the best loss with adding noise just to the instance of the pedestrian. It is interesting that such a simple method as horizontal flipping yields the best results. This is most probably caused by the fact that it enlarges the variance of the samples of the training set very well.

From the training learning curves shown in Figure 5.3, we can see that the baseline method trains much faster and has a higher loss and lower accuracy. However, this does not hold true for the validation accuracy and loss shown in Figure 5.4, where it can be seen that the loss of the baseline experiment is higher compared to the experiments using some augmentation and the accuracy of baseline experiment is lower.

To sum up, the performed experiments clearly show the benefits of using data augmentation for enlarging the training data set. The models trained with the use of such techniques yield better results and are able to generalize better to unseen data.

Chapter 6

Person image generator

The goal of the artificial neural network proposed in this chapter is to generate people in a given pose and in the given background. For this task, I experimented with multiple architectures based on the idea of Generative Adversarial Networks. First, I give an overview of the proposed network architectures in 6.1 and then describe more in detail the topology of generators and discriminators that I experimented with in section 6.1.1. In section 6.2, I continue with the enumeration of losses used for the training of the whole framework and then describe the training procedures in 6.3.

6.1 Proposed networks

The goal of this thesis is a data augmentation for neural networks training. As a domain for data augmentation, I chose the dataset that contains people in various positions and poses. However, these positions and poses do not cover the whole wanted spectrum of possible configurations.

As it was presented in the abstract, having the ability to generate people images in arbitrary, yet admissible, pose is a crucial prerequisite for Autonomous Driving applications. Firstly, because the existing datasets are quite limited in human pose variation and appearance. Secondly, because the strict safety requirements call for the ability of validation in rare situations. In other words, there is a need to have a possibility to sample from the huge distribution of admissible pose variants of pedestrians. Generating realistically looking people images is a very challenging problem due to various transformations of individual body parts [99, 100] self occlusions, etc.

In this section, I propose a novel approach for person image generation. This approach allows generating people images in a required pose, indicated by specific pose keypoints. It builds on top of the recent prevailing success of Generative Adversarial Networks described previously in the section 2.4. The contributions of this work comprise of the networks architecture, as well as the novel loss terms specifically designed to generate visually appealing

pedestrians seamlessly fitting the surrounding environment.

The framework consists of two essential components of each GAN - *generator* and *discriminator*. The goal of the generator is to generate realistic-looking images, and the discriminator goal is to provide a usable feedback about how good these images are. The structures of used generators are described in section 6.1.2 and of the discriminators in section 6.1.3.

There are two different topologies of the final framework that I experiment with. They mainly differ in the topology of the generator. Both generators and discriminators used are based on the ideas of progressive growing of GANs (section 2.4.3), conditional image inpainting (section 3.1) and person image generation (sections 3.5 and 3.6).

The first generator combines the encoder-decoder architecture used in [31, 29], and the second one is built upon the ideas of semantic image synthesis with spatially-adaptive normalization (SPADE) from [32].

For the purposes of the training of the proposed framework tackling this task, the dataset described in section 4.2 was created.

6.1.1 Topologies of generator and discriminator

Building blocks

There are three basic blocks in the networks - *upsampling* block, *downsampling* block and *SPADE residual* block. The first two blocks form the skeleton of both the *traditional* discriminator and the *encoder-decoder* generator and the third one of the *SPADE* generator.

Upsampling block. The upsampling block consists of two convolutional layers followed by an upsampling layer and can be found in generator only. Its input consists of the feature map obtained from the previous layer concatenated with an original input to the generator network. Having the original input injected into the input of each block should help the network to learn to preserve the background and modify the foreground (person) only.

The first convolutional layer has 3×3 filters with padding of size 1, stride 1, and reduces the number of filters by half.

The second convolutional layer has the same filter size, padding, and stride (as the first one) and keeps the number of input filters.

Each of the convolutional layers is followed by Leaky ReLU 2.2.3 activation with negative slope 0.2.

After the input is propagated through the convolutional layers and nonlinearities, it is upsampled by a bilinear interpolation layer.

To sum up, the layer takes an input of size $(c + c_{input}) \times w \times h$, where c is the number of channels from the previous block, c_{input} is the number of channels of the network input, w is the input width and h is the input height. The resulting output has a shape $c/2 \times 2w \times 2h$. This block is depicted in Figure 6.1.

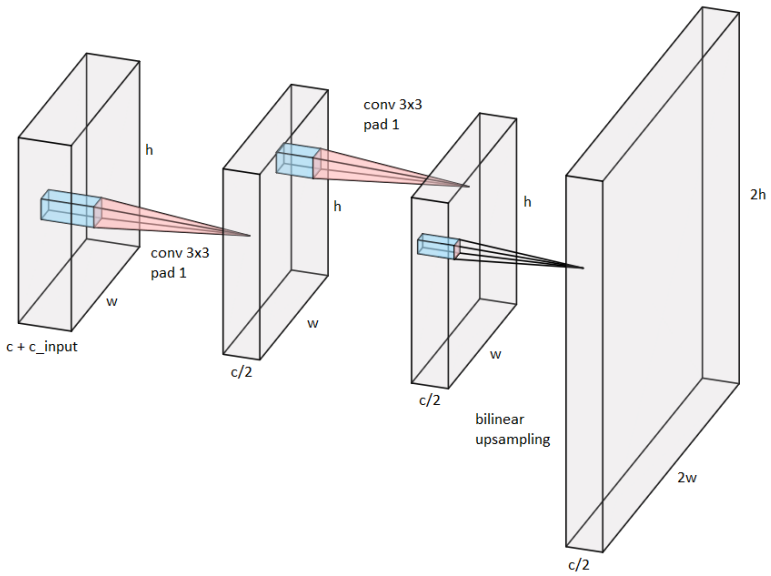


Figure 6.1: An example of the upsampling block. The input has c channels obtained from previous block concatenated with c_orig channels from the original input.

Downsampling block. The downsampling block can be found in the encoder part of the generator and in the discriminator. It consists of two convolutional layers, each followed by a nonlinearity, and of a downsampling layer.

The first convolution has a 3×3 kernel, padding of size 1 and a stride of 1 and keeps the number of the input channels from the previous block.

The second convolution has the same kernel size, padding, and stride as the first convolutional layer. The difference is that this layer doubles the number of the input channels.

Each of these layers is, just as in the upsampling block, followed by a nonlinearity.

These two convolutional layers are followed by a downsampling layer. The downsampling is performed by bilinear interpolation and reduces the spatial dimensions to a half.

To sum up the downsampling block, it takes an input of size $c \times w \times h$, where c is the number of channels, w is the width, and h is the height and outputs a tensor of size $2c \times w/2 \times h/2$.

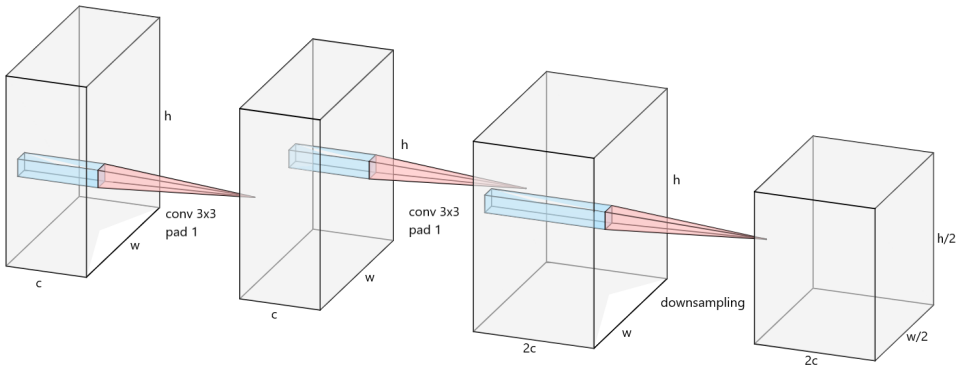


Figure 6.2: An example of the downsampling block.

SPADE residual block. *SPADE residual* block is used in the *SPADE* version of the generator. The structure of this residual block can be seen in Figure 6.3. It consists of two subblocks each with SPADE 3.4 normalization followed by ReLU activation and 3×3 convolution. The skip connection (dashed box) can also be learned with the subblock of the same structure. The number of channels reduces to a half after the pass through this block. Each of these residual blocks takes a different input resolution, and the conditional input to the SPADE normalization, therefore, needs to be downsampled to match it. The conditional input in the case of the proposed network consists of 17 channels that correspond to the location of 17 body joints and of 3 channels of the RGB background image.

6.1.2 Generator

Encoder-decoder generator

The generator has an encoder-decoder architecture with a *core block* between them. The overall structure of the generator can be seen in Figure 6.4. The convolutions in the generator were replaced by *gated convolutions* (section 3.1) in some of the experiments.

In the following text, there will be multiple topologies of different networks drawn. The legend to these figures is in Figure 6.5.

Encoder. The encoder is fed with an RGB image with a masked person. This input can be accompanied by additional information as keypoint locations or person shape. It reduces the spatial dimension of the input and increases the number of channels.

It is built from multiple donwnsampling blocks in order to arrive to the spatial resolution of $h_{core} \times w_{core}$ pixels which is an input to the *core block*.

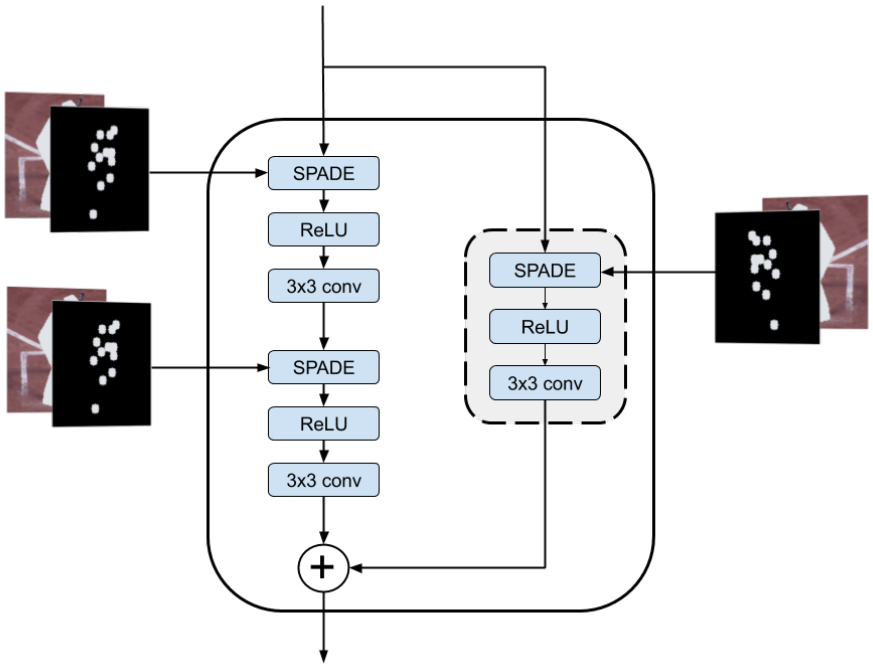


Figure 6.3: Topology of the SPADE residual block. The input corresponding to body joints are concatenated to one channel for better visualization.

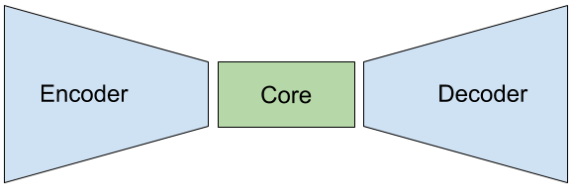


Figure 6.4: Overall structure of the generator.

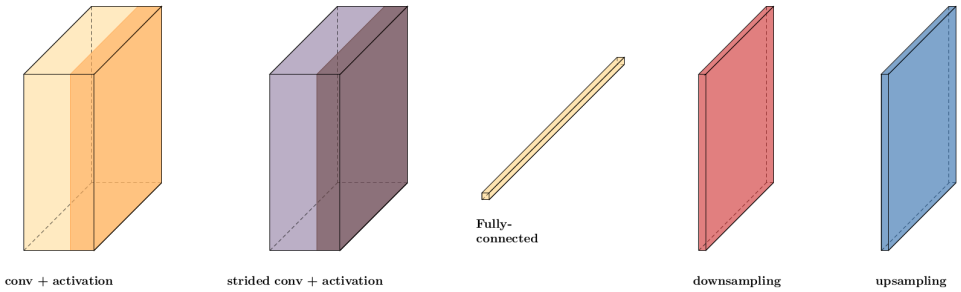


Figure 6.5: Legend to the networks.

Both width (w_{core}) and height h_{core} should be the same and must be a power of 2. Since each of the downsampling blocks reduces the spatial resolution by

half, the number of downsampling blocks can be computed as:

$$n_{down} = \log_2 \left(\frac{w_{in}}{w_{core}} \right) \tag{6.1}$$

where w_{in} is the input width that is the same as the input height h_{in}. The topology of the encoder is show in the Figure 6.6.

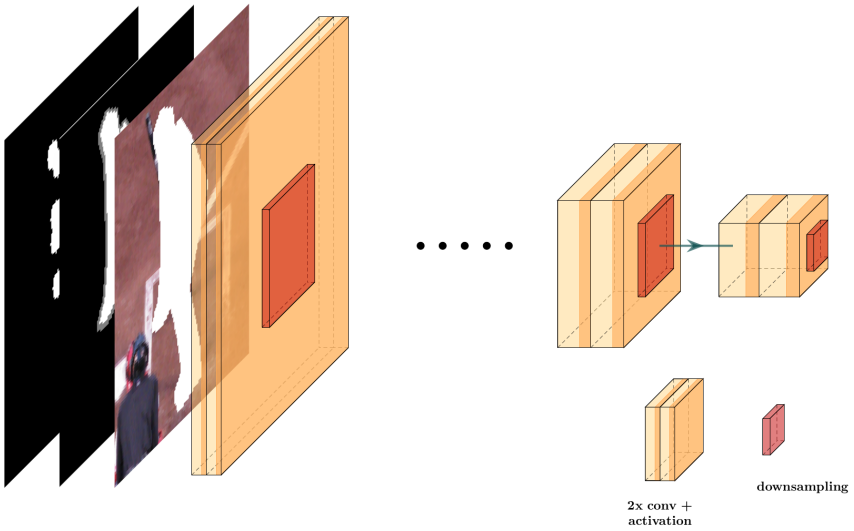


Figure 6.6: Encoder topology.

Core block. Core block forms a connection between the encoder and the decoder. It is inspired by the topology proposed in [23]. It consists of downsampling, residual and upsampling part - so it has a kind of encoder-decoder topology too.

The downsampling part consists of n_{down} blocks with each block consisting of a convolutional layer with 3×3 kernel, pad 1 and stride 1, followed by a 2×2 max pooling, normalization layer, and a nonlinearity.

The residual part consists of 3 residual blocks, each with a predefined number of subblocks. The only difference between those blocks is that the second block uses strided convolutions instead of normal ones.

The upsampling part is mirrored version of the downsampling part and therefore contains n_{down} blocks with each block consisting of upsampling layer, convolutional layer with 3×3 kernel, pad 1 and stride 1, normalization layer and a nonlinearity.

This block is followed by the last convolutional layer that takes as an input the output of the upsampling part concatenated with the downsampled network input. This should enforce the network to preserve the information about the background that should not be changed.

The topology of the core block is depicted in Figure 6.7.

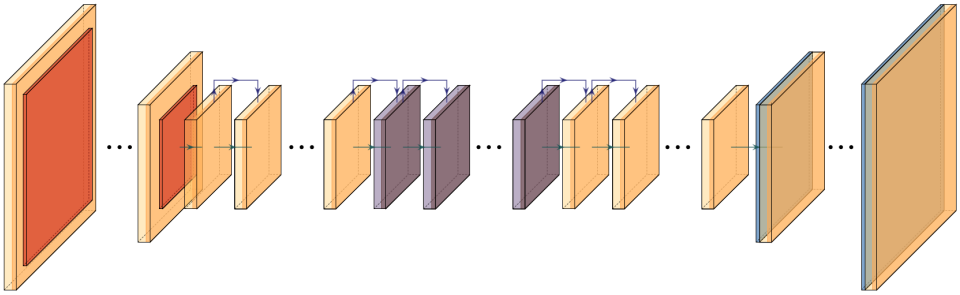


Figure 6.7: Core block topology.

Decoder. The decoder is a mirrored version of the encoder. It uses upsampling blocks instead of the downsampling ones. The number of these blocks is just the same as in the case of the downsampling blocks in the encoder. It takes output from the core block as its input and outputs RGB image.

It is built from multiple upsampling blocks. Its goal is to arrive at the spatial resolution of the original input (although this does not necessarily hold true in the case of progressive growing as it will be shown later). The topology of the decoder is shown in Figure 6.8.

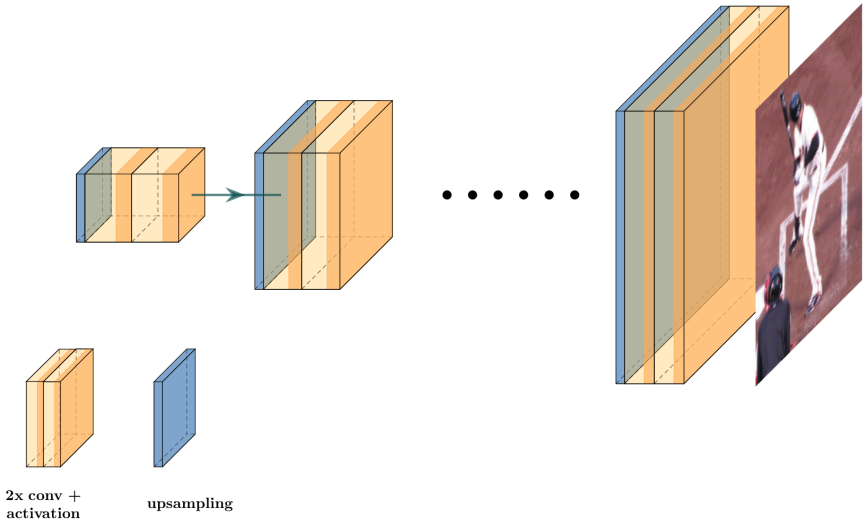


Figure 6.8: Decoder topology.

An example of the complete generator can be seen in Figure 6.9.

SPADE generator

In contrast to the previous *encoder-decoder* generator, the SPADE generator has no *encoder* part and therefore results in a more lightweight network. It takes a latent vector $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbb{I})$ just as in the original GANs. Note that there can be learned an additional encoder network that maps from a given

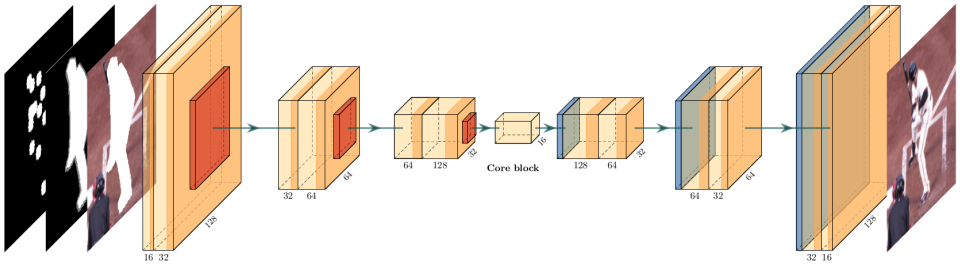


Figure 6.9: An example of generator that produces images up to 128×128 resolution. The generator takes as an input masked RGB image, mask of the person and keypoint locations.

input image of a person to $\mathcal{N}(\mathbf{0}, \mathbb{I})$. This allows to have more control over the generated person since the generation can be guided by this encoder.

At first, the generator takes $\mathbf{z} \in \mathbb{R}^d$ as an input, runs it through a linear layer and the output reshapes to $C \times 4 \times 4$, where C is the number of the channels, so it can be further passed to the convolutional layers. Then it applies n SPADE residual blocks each followed by a bilinear upsampling. This allows performing the progressive growing. The resulting image size s can be computed as

$$s = 4 \cdot 2^n \tag{6.2}$$

After these n upsampling blocks, the N resulting feature maps are passed to a 3×3 convolutional layer that maps it from N to 3 channels that correspond to R,G and B color channels of an image. Finally, a hyperbolic tangent (Tanh) activation is applied to this result in order to have the values in a valid range.

The main difference of this architecture in comparison to the *encoder-decoder* architecture is that the conditional input (background and keypoint locations) is passed to the network through SPADE blocks and is not fed to the generator as an input. The overall architecture of the SPADE generator is shown in Figure 6.10.

6.1.3 Discriminator

The goal of the discriminator is to provide feedback to the generator on how good the generated samples are.

The input to the discriminator is an RGB image. It is either the synthesized one from the generator or the real one from the dataset.

In this thesis, two discriminator architectures are tested. The first one resembles the discriminator from [25] and outputs only one number as traditional discriminators. The second discriminator architecture is often called PatchGAN.

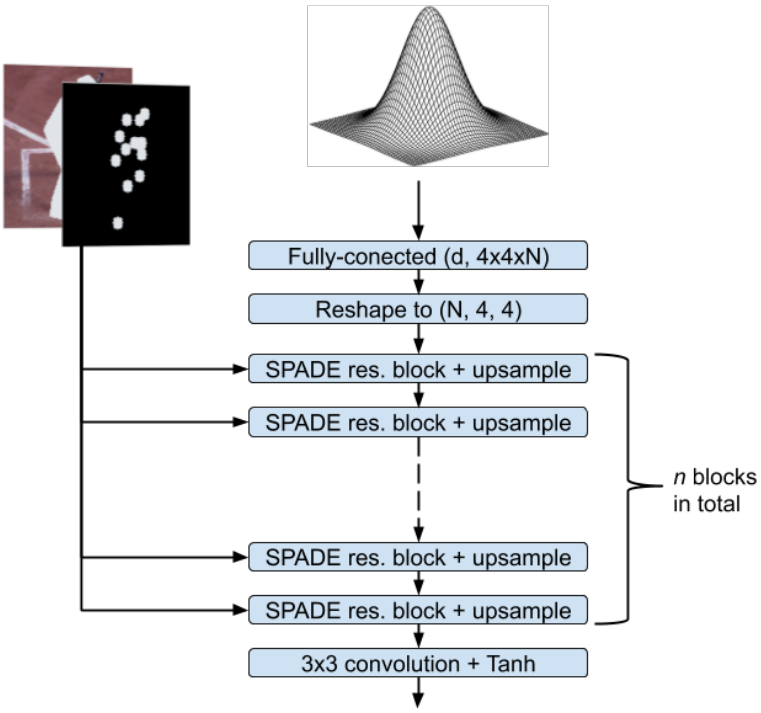


Figure 6.10: Diagram summarizing the architecture of the SPADE generator.

Traditional discriminator

The topology of the discriminator is quite similar to the topology of the generator decoder. It consists of multiple downsampling blocks and has an extra layer in the end that produces the final output number.

The downsampling blocks are used to reduce the spatial dimension to 4×4 pixels while increasing number of feature maps. After that, the intermediate result is fed into the final block.

The final block takes as an input n feature maps obtained from the previous block concatenated with an extra channel with values corresponding to the standard deviation of the input batch as in [25]. The intuition behind this mini batch standard deviation channel is that it should encourage the minibatches of generated and training images to show similar statistics. These $n + 1$ feature maps form the input to the final discriminator block.

In this final block, the input is first fed into 3×3 convolution with stride 1 and padding 1, which reduces the number of input feature maps by one to n. The second convolution has a 4×4 kernel, zero padding and stride 1. This means that the input of size $n \times 4 \times 4$ is transformed to the size $n \times 1 \times 1$. As an activation after both of these convolutions is used Leaky ReLU with negative slope of 0.2. Finally, to arrive at only one number per sample, a fully-connected layer is added on top of the convolutional layers. The final block is shown in Figure 6.11 with the complete discriminator in Figure 6.12.

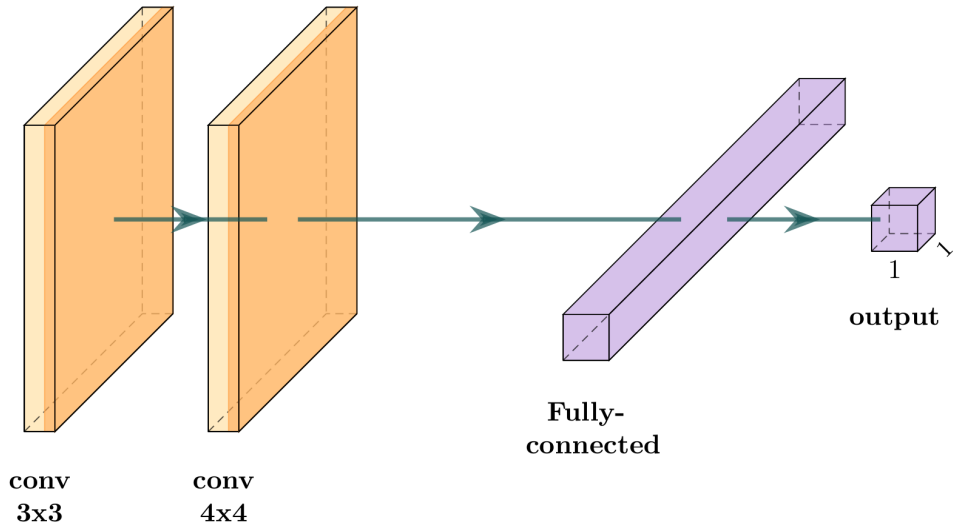


Figure 6.11: Topology of discriminator final block.

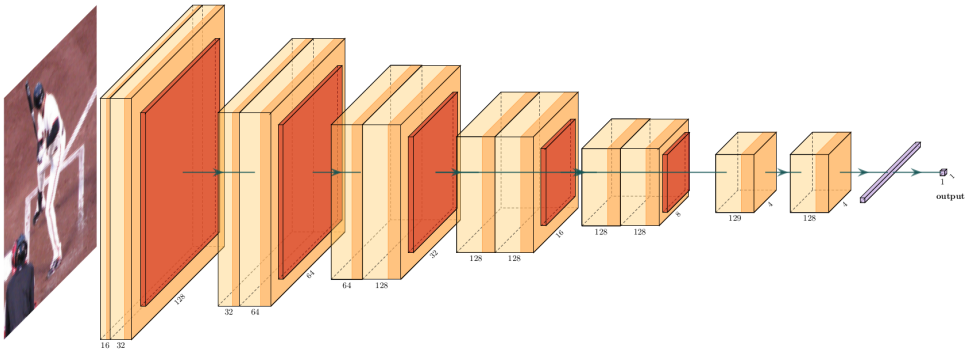


Figure 6.12: Complete topology of the basic discriminator.

The topology of the whole framework with both generator and discriminator network can be seen in Figure 6.13.

Patch discriminator

Patch discriminator (called Markovian discriminator or PatchGAN) was proposed in [73] where authors argue that the reconstruction losses such as L1 and L2 usually fail to encourage high frequencies but are often able to capture the low frequencies well. With the knowledge of this, it is not needed for the discriminator to enforce the low frequencies and it is desirable that it only models high-frequency structure well. Authors of [73] therefore propose to restrict the discriminator attention to the structure in local image patches only and design an architecture that only penalizes structure at the scale of patches. The goal of this discriminator is to classify each $N \times N$ image patch. This discriminator is run convolutionally across the input image, and

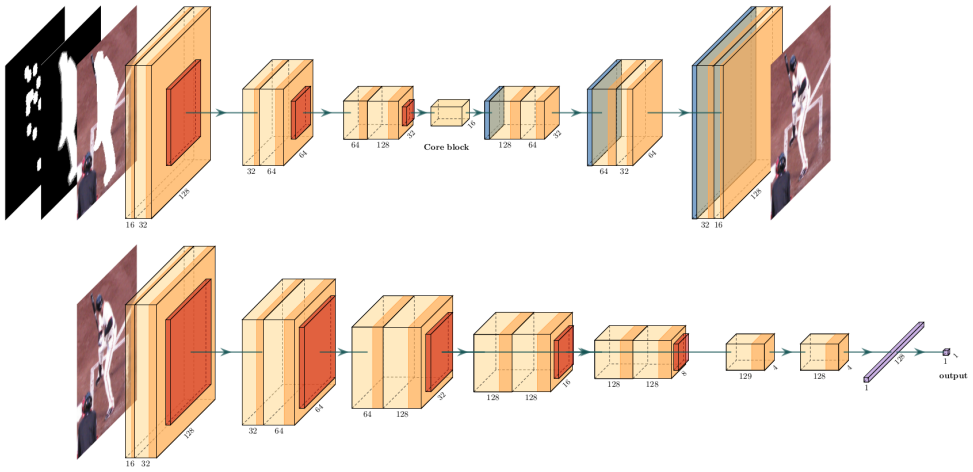


Figure 6.13: An example of encoder-decoder generator and traditional discriminator for the maximum size 128×128.

the responses are then averaged to provide the output.

The patch generator has n blocks, where n is the number of upsampling blocks in the generator, and each such a block consists of 4×4 convolution with stride 2 followed by a Leaky ReLU activation. After these n blocks, the resulting feature maps are passed to the Instance Normalization layer followed by Leaky ReLU activation and finally passed to a 4×4 convolutional layer that maps to just one output channel. This is shown in Figure 6.14.

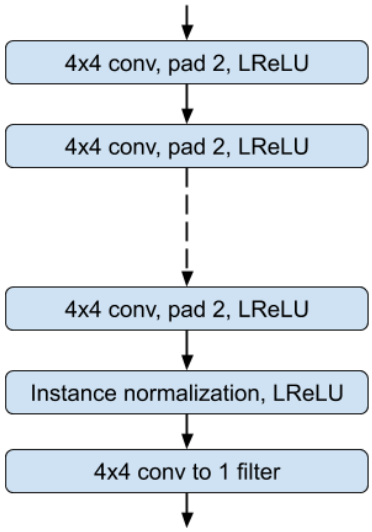


Figure 6.14: Architecture of a patch discriminator.

A traditional discriminator can be converted to patch discriminator quite easily. We can, for example, just replace the final discriminator block by a convolutional layer that will result in the one-channel output of responses.

6.1.4 Mask Estimation Network

Mask estimation network is used for predicting which pixels should be taken from the generated images and inserted to the original one. The input to this network consists of keypoint locations that are encoded in multiple channels. This network has a U-Net architecture (section 2.2.10). Since the resulting output image keeps on growing, this network has to grow too. It is done once again by progressive growing (section 2.4.3) of the upsampling (decoder) part. The output of this network is a one-channel mask $\mathcal{M} \in [0,1]^{H \times W}$, where (H, W) are the height and the width of the resulting image. Having the ability to output the values in the range of $[0, 1]$ brings the possibility to control borders of the painted person better and it should, therefore, result in more realistic-looking images than in the case of a simple crop of a generated person by the original mask.

Once the mask $\mathcal{M}$ is computed, it is used for merging the generated image I_G and input image I_{IN} into the resulting one I_{RES}. The resulting image is created as:

$$I_{RES} = \mathcal{M} \odot I_G + (I - \mathcal{M}) \odot I_{IN} \tag{6.3}$$

An example of the estimated mask based on the input that consists of keypoint locations can be seen in Figure 6.15.

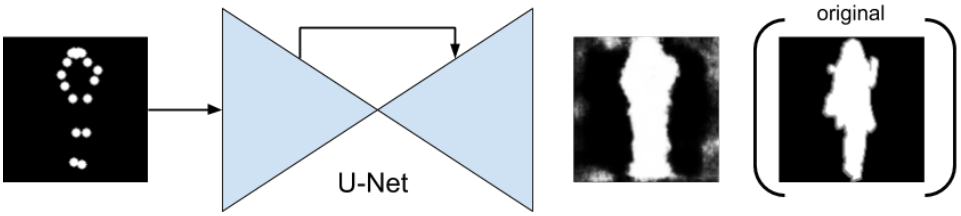


Figure 6.15: An example of mask estimation used for deciding which pixels take from the input image and which from the generated image.

You can see that the estimated mask does not precisely match the original one. However, one cannot expect it to do so since the network does not have any information about clothing and other things that can be included in the original mask. It serves mainly as a guidance to the generator in what pixels it should focus on.

6.1.5 Person style encoder

The random input to the SPADE generator should provide a way for multi-modal synthesis. However, this input does not need to be completely random. One can attach an image encoder that will encode the style of a given person and map it to a normal distribution. This style encoder and SPADE generator therefore form a variational autoencoder [101] where the encoder captures

the style of the input person, and the generator then combines encoded style and conditional input to the final image.

This encoder is implemented as a CNN with n blocks of 3×3 convolutions with stride 2 followed by an instance normalization and LReLU activation. These n convolutional blocks are followed by two fully-connected layers which produce the mean and variance of the output distribution as in [32]. The topology of such a network is shown in Figure 6.16.

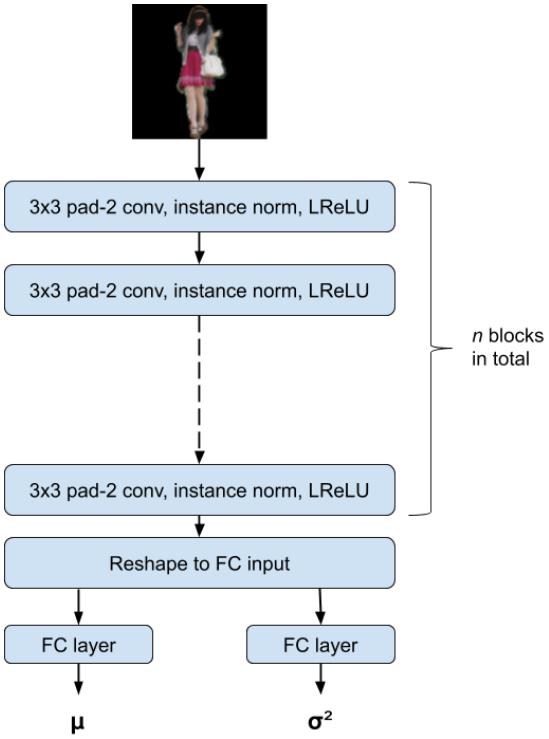


Figure 6.16: Topology of the style encoder..

6.2 Used losses

Wasserstein loss. Wasserstein loss is one of the most commonly used losses in GAN training. This loss is previously described in section 2.4.2. It is used in this thesis for its good results and stability.

Identity loss. Identity loss enforces the generator to generate similar images to the target ones. In experiments, this loss is implemented as either the $L1$ and $L2$ loss 6.2. There can be multiple identity losses in the training setup. One can, for example, have $L1$ loss that enforces the generator not to change the background and then $L2$ loss that only focuses on the foreground (painted person).

Feature matching loss. Feature matching loss is yet another commonly used loss in the GAN training. It aims to match the features obtained from some network. In the experiments, this network was chosen to either be the discriminator network or the VGG network or both.

LBP loss. The *LBP* loss (mainly the *soft* version) is a novel (to the best of my knowledge) loss that I propose in this thesis. It should enforce the generated input and ground truth input to have similar edges. This is done by computing the difference of the LBP descriptors computed on the original person and on the generated one. This difference can be, for example, computed with $L1$ loss, which is equal to the computation of Hamming distance.

The drawback of the computation with the LBP features 2.5.1 is that the process of computing such features is not differentiable due to the thresholding used in the computation.

To tackle this, I propose a *soft* version of LBP. The feature vectors of such *soft* version are obtained simply by taking the 8-dimensional vector just before the thresholding in the computation of traditional LBP. Therefore, in the soft version, we obtain for each pixel a feature vector $f \in [-1, 1]^8$ which corresponds to the image gradient in that pixel instead of a decimal number as in classical LBP. When using this version, the computation is fully differentiable since no thresholding is used.

The most significant advantage of this loss is that it does not by itself try to preserve original colors as other identity-preserving losses do. It only keeps the local information about the image gradient. This implies that the results obtained when trained with this loss should generalize better than when using traditional identity losses such as L1 or L2 loss.

The *soft* version can be efficiently implemented with the use of convolutions. The convolution filter is of size $1 \times 8 \times 3 \times 3$ and maps from a gray-scale image (1 channel) to an 8-D feature vector that corresponds to the image gradient in a given pixel. Each of those 8 kernels of the filter has -1 in the center and exactly one 1 in the rest. Convolving the input grayscale image with such filters results in a *soft* version of LBP. These filters are shown in Figure 6.17.

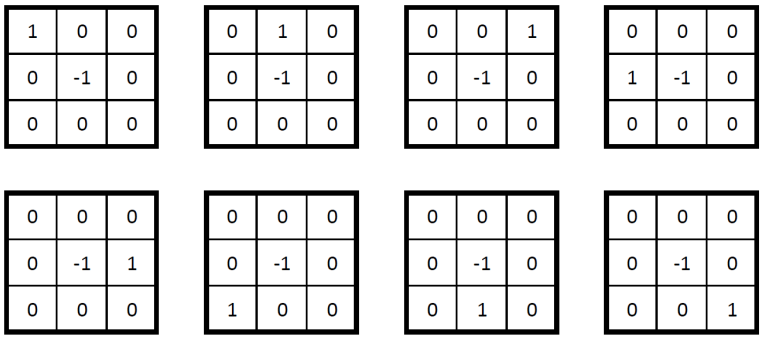


Figure 6.17: Filters used for computation of *soft* LBP.

Illumination loss. The illumination loss is another newly proposed loss in this thesis. It is a loss based on the difference of image histograms (section 2.5.2). The image histograms can be computed over various inputs, in this work are used these

- *grayscale* image - it is a common choice to compute image histogram over the grayscale image
- *value* component of the HSV color model (sec. 2.5.3)
- *lightness* component of the HSL color model (sec. 2.5.3)

Once the histograms of the original image and of the generated one are computed, they are compared by the use of one of the distance measured shown in section 2.5.2.

The intuition behind this loss is that the original and generated person should obey the lighting conditions of the image. If this was not taken into account, there might be a very bright person generated to a very dark scene which might not seem natural.

KLD loss. Kullback–Leibler divergence (KLD, section 2.4.1) is used to estimate how much does one distribution differs from another. In neural networks, this is commonly used in such cases when one wants to model a Gaussian distribution, e.g., in Variational Autoencoders [101, 102].

This loss can be written as:

$$\mathcal{L}_{\text{KLD}} = \mathcal{D}_{\text{KL}}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})) \tag{6.4}$$

where $p(\mathbf{z})$ is a prior distribution chosen to be a standard Gaussian distribution and and the variational distribution q is fully determined by a mean vector and a variance vector [101] outputed by Style Encoder 6.1.5 [32].

To allow the gradient backpropagation to the Style Encoder, the reparametrization trick [101] is used.

6.3 Training procedure

In this section, I will describe the techniques used for the training of the proposed framework and the overall training procedure. The training is based on the progressive growing proposed in the [25], which includes growing of the input/output resolution, layer blending, and a few other techniques.

6.3.1 Progressive growing

The idea of progressive growing was described in section 2.4.3 and is used in training. With the generator, the progressive growing is either used in both encoder and decoder or in decoder only. In the discriminator, it is used always.

As it can be seen in Figure 6.9, the generator does not necessarily output 3 channels that would correspond to the RGB channels. The same holds for the discriminator (Figure 6.12), where there might be more than 3 channels required in the input layer. To tackle this issue, temporary layers that either match from multiple feature maps to RGB (called *toRGB*, used in generator) and from RGB to feature maps (called *toFeatures* or *fromRGB*, used in generator encoder and in the discriminator) are used. These layers are implemented by 1×1 convolutions which correspond to either reducing or enlarging the number of feature maps.

In the setup of the experiment, it is required to specify the number of epochs of the training for each resolution together with a fade-in percentage that controls the transition to the higher resolution.

The fade-in percentage specifies the number of epochs when the output from the previous resolution will be upsampled and blended with the output of the newly trained layer. This procedure can be seen in the example in Figure 6.18. The figure shows the decoder part of the generator and the discriminator in the case of growing the resolution from 16×16 to 32×32. In (a) the initial setup before the resolution growth is shown. In (b) the transition is shown. During this transition, the layers that work on higher resolutions are treated like residual blocks, whose weight α linearly increases from 0 to 1 during the predefined number of epochs. The blocks $2\times$ refers to doubling the resolution, and $0.5\times$ corresponds to halving the input resolution. Finally, you can see the newly blended-in layer after the transition phase in (c).

6.3.2 Normalization and weight scaling

The used normalization and weight scaling during the runtime is motivated by the work [25].

The dynamic scaling of weights is performed by setting $\hat{w}_i = w_i/c$, where w_i are the weights and c is the per-layer normalization constant from He's initializer [103, 25]. In the case when some parameters have a larger dynamic range than others, they will take longer to adjust. The benefit of performing the weight scaling dynamically is that the dynamic range, and therefore the learning speed too, is the same for all weights.

It is often needed to use some kind of normalization on order to avoid the magnitudes in generator and discriminator to blow out. In order to avoid

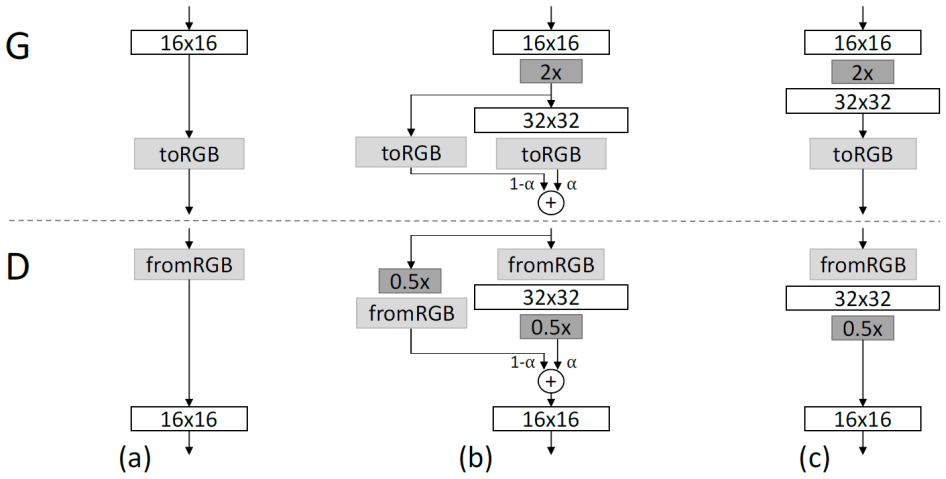


Figure 6.18: An example of the procedure of adding a new blocks to current networks and therefore increasing the resolution [25]. G denotes the decoder part of the generator and D stands for the discriminator.

this, the authors of [25] propose a *pixelwise normalization* that is a variant of local response normalization [64]. The formula describing this normalization is written in the equation 6.5 where $a_{x,y}$ and $b_{x,y}$ are original and modified feature vector at spatial position (x, y) and $\epsilon = 10^{-8}$ is an additional constant term used for numerical stability.

$$b_{x,y} = \frac{a_{x,y}}{\sqrt{\frac{1}{N}\sum_{j=0}^{N-1}\left(a_{x,y}^{j}\right)^{2} + \epsilon}} \tag{6.5}$$

The authors claim that with the use of this normalization, the escalation of signal magnitudes is effectively prevented. Pixelwise normalization is used in the generator in this work.

6.3.3 Nearest Neighbor Search

A question of whether the generator will be able to generalize well arises when it is trained only to paint one possible person per training sample.

To tackle this issue, I propose to alternate between multiple persons that can be painted in the image with the given pose. This requires to find such a person that would have a pose similar to the pose of the original person.

A proposed solution therefore performs the nearest neighbor search with respect to the pose (skeleton) given by keypoint locations.

First, all the skeletons needed to be normalized. This is done by centering all the skeletons to the center of their torso. This center, denoted as $\mathbf{t}$, is computed as mean of the point in between the shoulders and of the point

between the hips:

$$\begin{aligned} \mathbf{t}_{hips} &= \frac{1}{\sum\limits_{i \in id_{hp}} v_i} \sum_{i \in id_{hp}} \mathbf{x}_i \cdot v_i \\ \mathbf{t}_{shoulders} &= \frac{1}{\sum\limits_{i \in id_{sh}} v_i} \sum_{i \in id_{sh}} \mathbf{x}_i \cdot v_i \\ \mathbf{t} &= \frac{1}{2} \left(\mathbf{t}_{hips} + \mathbf{t}_{shoulders} \right) \end{aligned} \tag{6.6}$$

where $\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_N)$ are keypoint locations $\mathbf{x}_i = (x_i, y_i)$ of a skeleton, $\mathbf{v} = (v_1, ..., v_N), v_i \in \{0, 1\}$ is a visibility variable with 0 denoting invisible keypoint and 1 denoting a visible one and id_{hp} and id_{sh} are indices from $(1, ..., N)$ that denote the indices of hips and shoulders in $\mathbf{x}$ and $\mathbf{v}$. N denotes number of all possibly annotated keypoints.

Second, the skeletons are rescaled. This is done by enforcing that the height of the torso h_t takes a predefined ratio r_t of the person height. The torso height is computed as

$$h_t = \mathbf{t}_{hips}(y) - \mathbf{t}_{shoulders}(y) \tag{6.7}$$

where $\mathbf{t}_{hips}(y)$ and $\mathbf{t}_{shouders}(y)$ denote the y-coordinate of the middle point of hips and shoulders respectively. Having the height of the torso, the scale s is computed as

$$s = \frac{h_t}{r_t} \tag{6.8}$$

The transformed keypoint locations $\hat{\mathbf{x}}$ are obtained with the computed torso center $\mathbf{t}$ and the scale s from the original keypoint locations $\mathbf{x}$ as

$$\hat{\mathbf{x}} = s \cdot (\mathbf{x} - \mathbf{t}) \tag{6.9}$$

Afterward, a feature vector for skeleton $\hat{\mathbf{x}}$ is computed. These features are computed similarly to the approach proposed in [104]. The feature vector is computed with the Algorithm 2, where E is a set of index pairs of keypoints

that are connected to form the skeleton.

Result: A feature vector $\mathbf{f}$
initialize $\mathbf{f}$ to be an empty vector;
for $(i, j) \in E$ **do**
  if $v_i > 0$ *and* $v_j > 0$ **then**
    $\mathrm{d}x = |x_i - x_j|$;
    $\mathrm{d}y = |y_i - y_j|$;
    $\mathbf{f}_{i,j} = (\mathrm{d}x, \mathrm{d}y, \mathrm{d}x^2, \mathrm{d}y^2)$;
  else
    $\mathbf{f}_{i,j} = (0, 0, 0, 0)$
  end
  append $\mathbf{f}_{i,j}$ to $\mathbf{f}$
end

Algorithm 2: An algorithm for computing a feature vector from keypoints of the skeleton.

This algorithm gives us a fixed-size vector $\mathbf{f} \in \mathbb{R}^{4|E|}$, where $|E|$ denotes the number of all possible connections in the skeleton, for each skeleton regardless of the number of annotated keypoints.

Once we have such a feature vector for all the skeletons from the dataset of size M, we stack them to the matrix $\mathbf{F} \in \mathbb{R}^{4|E| \times M}$. We can now query any skeleton represented by a feature vector $\mathbf{f}_q$. The distance used for comparison of the feature vectors is chosen to be a squared Euclidean loss. With the use of matrix $\mathbb{F}$, this can be simply done as

$$\text{score} = (\mathbf{F} - \mathbf{f}_q)^T (\mathbf{F} - \mathbf{f}_q) \tag{6.10}$$

and then taking the top score in order to obtain the nearest skeleton in the dataset (note that in the case that the query feature vector is in the dataset, the nearest skeleton will always be the query one and should be therefore discarded). The operation $\mathbf{F} - \mathbf{f}_q$ means subtracting the vector $\mathbf{f}_q$ from each column of the matrix $\mathbf{F}$.

Another way of obtaining scores is to measure the cosine distance between the query feature vector and the column vectors of the matrix $\mathbf{F}$.

Despite the simplicity of this approach, it yields good results as it can be seen in Figure 6.19.

Using this technique, it is possible to find a person in the dataset that has a similar pose to the ground truth person and can therefore be used quite well for the training. It is also possible to retrieve not just the nearest neighbor but also k nearest neighbors. One can then choose to for example alternate between 4 possible persons that should be generated in a given pose.

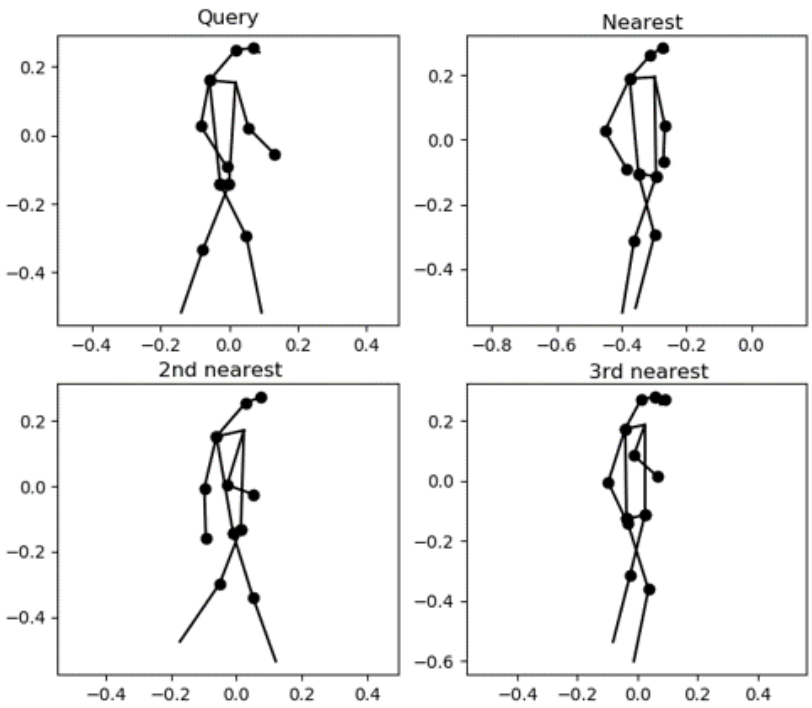


Figure 6.19: An example of nearest neighbor search among the skeletons.

Chapter 7

Experiments with person image generator

In this chapter, I will show results of pedestrian generation of the experiments described in the previous chapter. In section 7.2, I show the generated samples and discuss the results from both experiments. In the following sections, I discuss and evaluate only the results of the experiment with the SPADE generator since it yields better results. I give an analysis of failure cases in 7.2.3, show results of human evaluation of generated samples in 7.3 and show results of the human detector on generated images in comparison to the real ones in section 7.4. Furthermore, in section 7.5, I perform experiments with augmenting an existing dataset with images generated with the proposed method.

In order to be fair, I fixed the samples that will show here for a better comparison. Note that these samples are randomly drawn from the validation set and are not cherry-picked in order to show just a few nice results.

7.1 Network setup

In this section, I will show the setups and results of some of the performed experiments. Each experiment will be described by two tables. The first table contains the general setup of the experiment such as the maximal resolution of the generated person, number of layers, and whether gated convolutions were used. An example of such a table can be seen in table 7.1 along with the explanations. The second table provides an overview of used losses along with the information whether they were computed over the background, foreground, or over the whole image. This table also shows in which stages of the training was each loss used. An exemplary table with explanations can be seen in 7.2.

I will show the results of one experiment with *encoder-decoder* generator and one experiment with *SPADE* generator. Note that many more experiments were performed but only those with best results with each one of these generator architectures are shown.

feature	*setting*	*(explanation)*
generator	SPADE	Which generator is used. Either *encoder-decoder* or *SPADE*.
discriminator	patch	Which discriminator is used. Either *traditional* or *patch*.
starting resolution	16x16	Starting (minimal) resolution of the progressive growing.
maximal resolution	64x64	highest generated resolution
number of up/downsampling layers	2	number of growings of the generator and the discriminator
blend-in percentages	X,X,X	Fraction of epochs during which undergoes the transition to a higher resolution. First value is not used (since first layer is not blended).
Dynamic weight scaling	generator	If and eventually where is dynamic weight scaling used
gated convolutions	✓	Shows whether the gated convolutions were used.
learning background/mask	×	Shows whether the person was just cropped according to the ground-truth mask or whether this region was learned.

Table 7.1: An example of a table with an experiment setup and explanations.

loss	*region*	*depths*			*(explanation)*
		0	1	2	
L1	foreground	✓	✓	✓	Name of the loss, region where this loss is applied, training stages where this loss is used.

Table 7.2: An example of a table with used losses and explanation.

7.1.1 Experiment with encoder-decoder generator

In this experiment, I use the encoder-decoder generator and a traditional discriminator. The complete setup of the network is shown in table 7.3. The overall topology of the network is similar to the one in Figure 6.13 except for the dimensions.

As the reconstruction loss used to train the generator, I use the edge (soft

feature	*setting*
generator	enc.-dec.
discriminator	traditional
starting resolution	8x8
maximal resolution	64x64
number of up/downsampling layers	4
blend-in percentages	x,20,20,20
Dynamic weight scaling	generator
gated convolutions	✓
learning background/mask	×

Table 7.3: Setup of the experiment with encoder-decoder generator.

LBP) loss (section 6.2) with L1 distance used to measure the difference of the features obtained from the ground-truth image and from the generated one. As a GAN loss, I used a masked version of WGAN with gradient penalty. The setup of losses is summarized in table 7.4.

loss	*region*	*depths*			
		0	1	2	3
WGAN-GP	foreground	✓	✓	✓	✓
edge loss	foreground	✓	✓	✓	✓

Table 7.4: Losses used in the setup with encoder-decoder generator.

7.1.2 Experiment with SPADE generator

This experiment uses *SPADE* generator and *patch* discriminator. Both of these networks are trained in the progressive fashion from the resolution 8×8 pixels all the way to the final resolution of 128×128 pixes. The complete setup is shown in table 7.5. The conditional input to the SPADE layers consists of keypoint masks and of the masked background image.

Used losses are shown in table 7.6. In this experiment, the only reconstruction loss used is the edge loss based on the soft LBP proposed in section 6.2. With the use of this loss, I hope to achieve better results in terms of image quality and diversity. As a GAN loss, I used WGAN with gradient penalty.

On top of these losses, two feature matching losses are used. The first one measures the L1 distance of discriminator features obtained with the real and the generated samples. The second one compares the features in a similar

feature	*setting*
generator	SPADE
discriminator	patch
starting resolution	8x8
maximal resolution	128x128
number of up/downsampling layers	5
blend-in percentages	x,50,30,20,15
Dynamic weight scaling	none
gated convolutions	×
learning background/mask	✓

Table 7.5: Setup of experiment with SPADE generator.

fashion but obtains them from a pretrained VGG19 network. The benefits of using pretrained network to get features for matching compared to the ones trained from scratch are mainly at the beginning of the training. In the early stages, the discriminator is not able to discriminate well between the real and fake samples and therefore its features for real and fake samples do not differ that much. However, this is not the case when using pretrained network.

loss	*region*
discriminator feature-matching	foreground
VGG feature-matching	foreground
Edge loss (soft LBP)	foreground
WGAN-GP	all

Table 7.6: Losses used in setup with SPADE generator.

The topology of the whole framework that captures the training procedure is shown in Figure 7.1. It consists of 3 basic parts:

1. *Generator and discriminator*: building block of the GAN setup (details in Figures 6.10 and 6.14),

2. *image encoder*: a convolutional neural network outputting the parameters μ and σ^2 of the normal distribution (detail in Figure 6.6) and

3. *mask estimation network*: U-Net network (detail in Figure 6.15) that has keypoint masks as input and outputs an estimation of the mask.

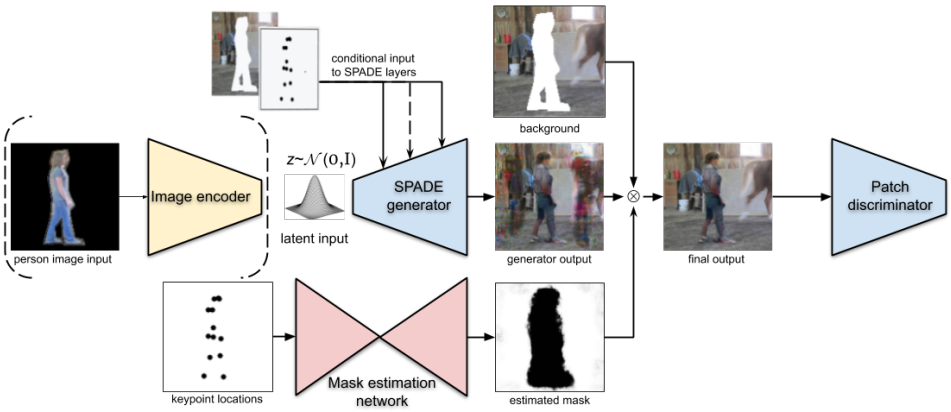


Figure 7.1: Topology of the framework with SPADE generator. The blocks drawn in blue are traditional parts of GANs. The image encoder drawn in yellow is a convolutional neural network and mask estimation network drawn in pink has a U-Net architecture.

7.2 Visual evaluation

7.2.1 Encoder-decoder generator

In Figure 7.2 are shown examples of images generated by the encoder-decoder generator in comparison to the original ones. It is clear that despite the fact that at some images (e.g., in the third row) the network managed to produce quite good results, the overall image quality is low. The reasons behind these poor results can be in the fact that all the information is fed to the network at one input and it is difficult for the network to preserve the information about the pose from the input all the way to the output.

The results obtained by using SPADE generator (as shown in the next section) are superior to results from the encoder-decoder generator. This is the reason why further evaluations are only with the samples generated with SPADE version.

7.2.2 SPADE generator

Figure 7.3 shows a comparison of the original samples and final samples generated with the SPADE generator. The results are clearly superior to the results obtained with the encoder-decoder generator. One of the reasons for this is that with the use of SPADE layers, the network is able to focus more on the given input pose since it is fed to the network at multiple points.

In the upper left pair, we can see that the network is able to generate even the faces quite well. However, it still has problems with drawing the clothing consistently over the whole person. In the middle pair in the second row

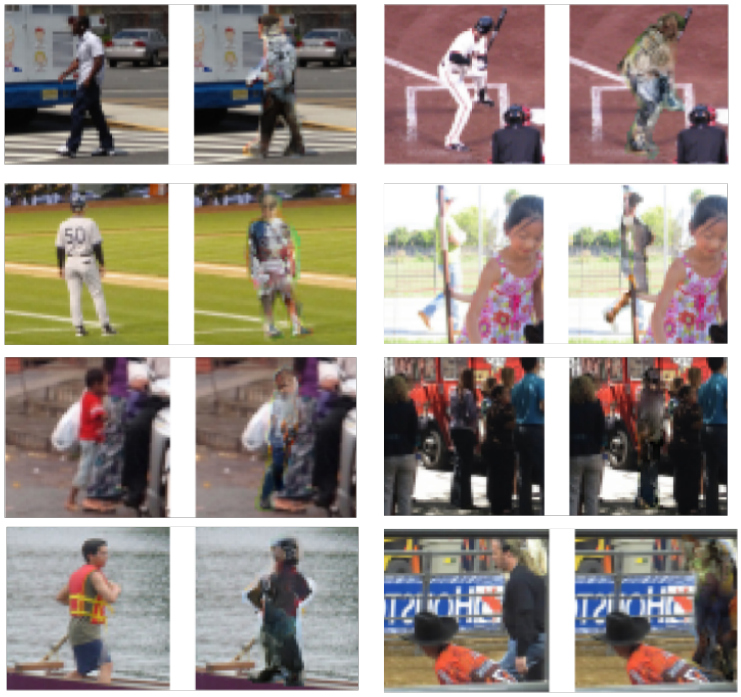


Figure 7.2: Comparison of ground-truth (first and third row) and generated images by the encoder-decoder generator.

and in the first pair in the last one, we can see that the network generated humans that are standing with their backs to the camera. This demonstrates its ability to incorporate the input pose information to the final result.

In these two cases, there is an occlusion of the generated person, and the network is able to keep the objects occluding the person. This is possible due to the fact that the conditional input consists of the background, and the network was able to learn to redraw only the foreground pixels corresponding to the chosen person.

In Figure 7.4, we can see ground-truth masks compared to the masks estimated from the conditional input that consists of keypoint locations. We can see that the network generating the mask is able to estimate it quite well. This mostly helps to achieve a more natural blending of the generated image to the original one. This network is, of course, not able to capture the overlapping objects as it can be seen in some of the examples. However, the most interesting part is that this network is able to generate these mask estimation without any loss that would be straight used to train it.

7.2.3 Analysis of failure cases

I show and describe a few of the common failure cases that can be seen in Figure 7.3 shown above. From left to right I will show the result of the

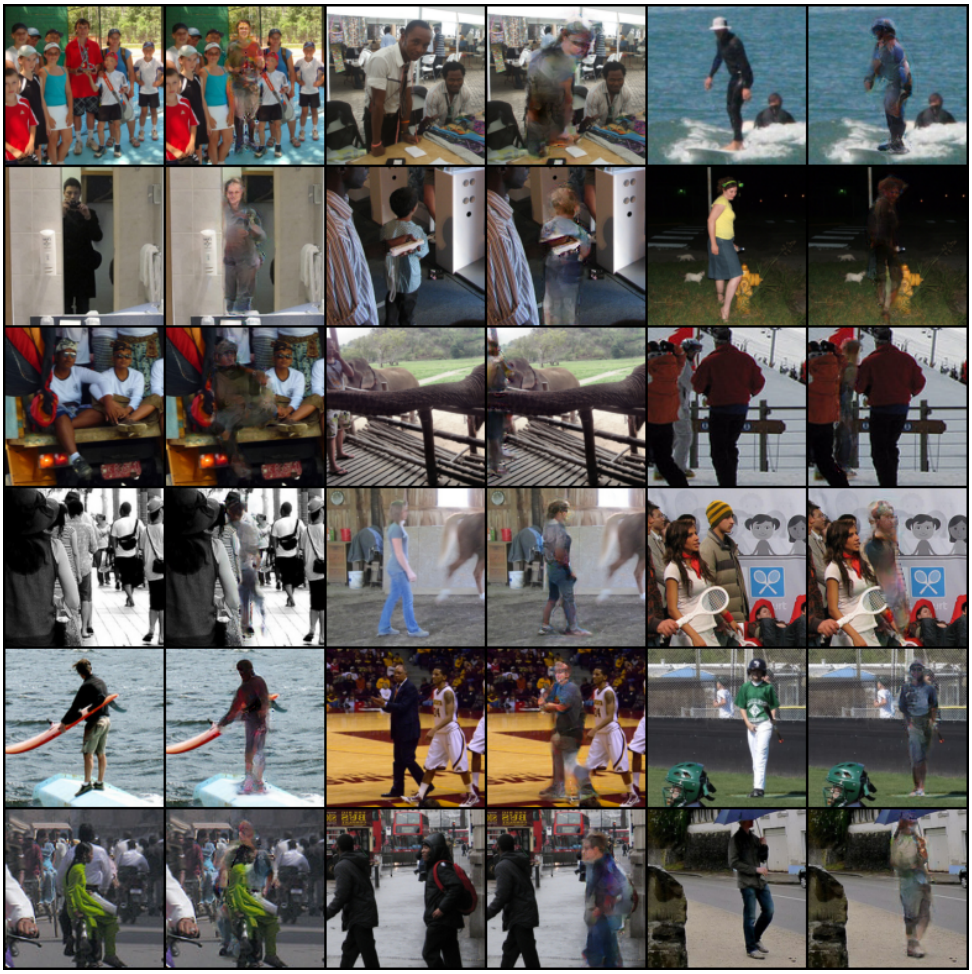


Figure 7.3: Comparison of the original samples with the samples generated with the proposed framework.

generating part of the framework, result of the mask estimator, final blending to the original image and the ground-truth image.

In Figure 7.5 is shown the case of inconsistently drawn clothing. In the first row, the bottom of the left leg of the generated person is not covered with the trousers despite the fact that the right one is. One might say that it can be some kind of style, but it most surely was not intended. The example shown in the second row has a similar problem with the left leg where there is an occlusion with the right leg of the player standing nearby, and the network does not manage to capture this well. This degenerates the overall generated image considerably since the upper half of the generated person looks pretty well.

A failure case related to a rare pose can be seen in Figure 7.6. In this example, the pose of the original person is pretty non-standard and therefore there are not many examples of people in such poses in the training set. This results in a poorly generated image where there is no big variance in the color

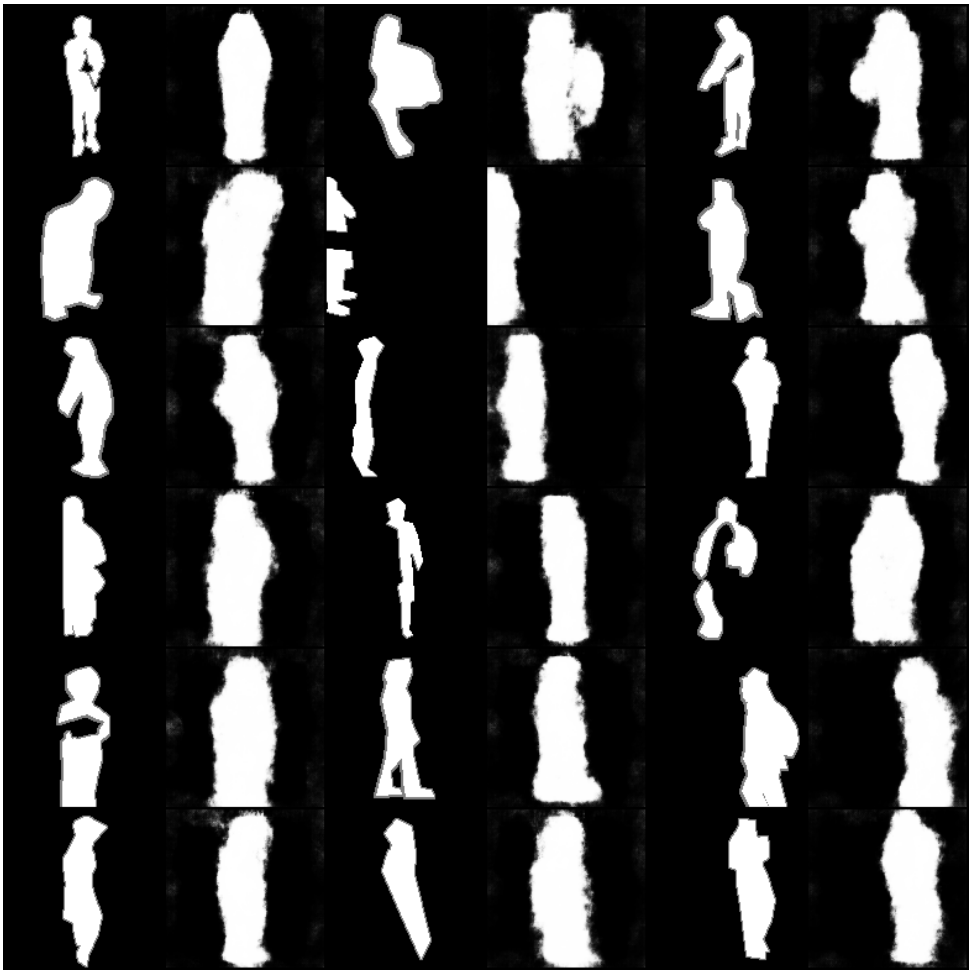


Figure 7.4: Comparison of the original samples with the samples generated with the SPADE generator.

Figure 7.5: An example of the inconsistent clothing and wrongly handled occlusion.

of the clothing. However, one can observe that the network managed to draw

a quite good face and that it almost succeeded in drawing crossed legs.

Figure 7.6: An example of the generation in rare pose.

7.3 Human evaluation

Two different user studies with surveys were done. The goal of the first one is to rank the quality of the shown image and the goal of the second one is to compare the quality of a generated image with respect to the original image.

7.3.1 Image quality ranking

The first survey asks the participants to rank the quality of the shown image on the scale from 1 to 10 where 1 is completely unrealistic image and 10 is the image indistinguishable from the real one. There are 18 images in total out and half of those are fake images generated with the proposed framework. This survey was filled by 150 volunteers. The results of this survey are shown in Figure 7.7 with the mean and standard deviation of the ranking of both original and generated samples in table 7.7.

input	*mean*	*standard deviation*
original	7.732	0.612
generated	3.417	1.180

Table 7.7: Mean and standard deviation of ranking of original and generated images.

In Figure 7.8 are shown the best ranked generated images and the worst ranked real images.

7.3.2 Comparison of real and generated images

The second survey asks the volunteers to compare the original and the generated image and give points according to which image they prefer. The scale of the possible values is once again from 1 to 10 where 1 and 10 mean

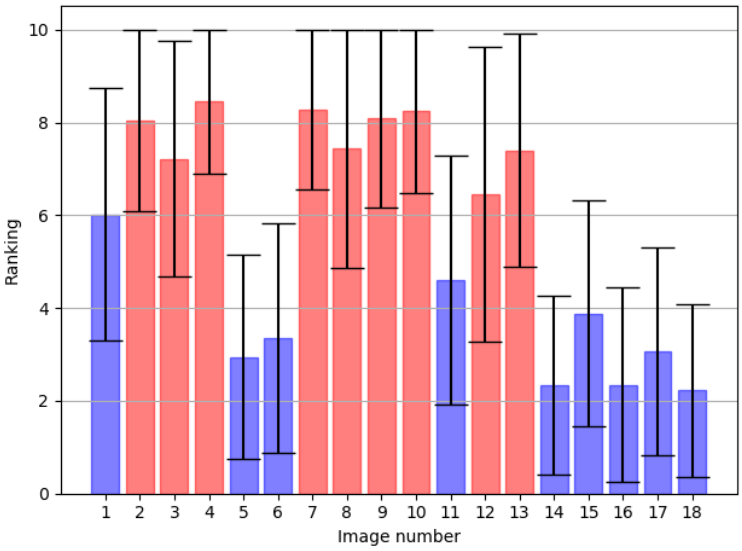


Figure 7.7: A bar plot showing an average ranking and stadard deviation of real samples (in red) and of generated samples (blue).

Figure 7.8: From left to right: two best generated images, two worst real images.

that one image is strictly better and the other one is complete nonsense. In total, 106 people participated in this survey and compared 12 given image pairs. The results are shown in Figure 7.9.

In Figure 7.10 are shown the image pairs where the volunteers preferred the generated images the most.

From both of these surveys, it is clear that the generated samples are yet not good enough to fool humans. However, the results are not completely pessimistic and give a lot of room for improvement.

7.4 Human detector performance

In order to test how well are the generated samples detected by the object detector, I run YOLOv3 detector [105] on images from the validation set. First I run the detector with real humans and then compare it with the results

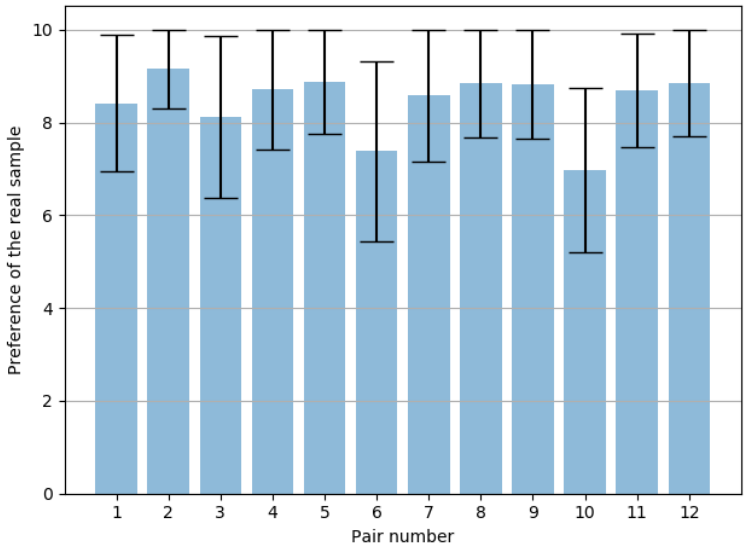


Figure 7.9: A bar plot showing the preference of the real sample over the generated one.

Figure 7.10: Two most preferred generated images (in pairs on right) with their real counterparts.

of this detector on the generated images.

The detections are compared by the Intersection over Union of the ground-truth bounding box and the detected one. Comparisons of detections can be seen in Figure 7.11. The first row shows examples where the detector works relatively well when compared to its performance on the original image. The first pair on the second row is very interesting in the fact that the detector was not able to detect the person on the original image, but it detected the generated one which does not look very appealing to a human eye. Finally, the last pair shows an example when a badly generated person is not even detected.

The average IoUs for original and generated images are shown in table 7.8. It is clear that the detector achieves better results on the original images. However, the mean IoU on the generated person images is not very low when compared to the mean IoU on the originals. It follows that the generated images can serve for the augmentation of the current datasets well even at

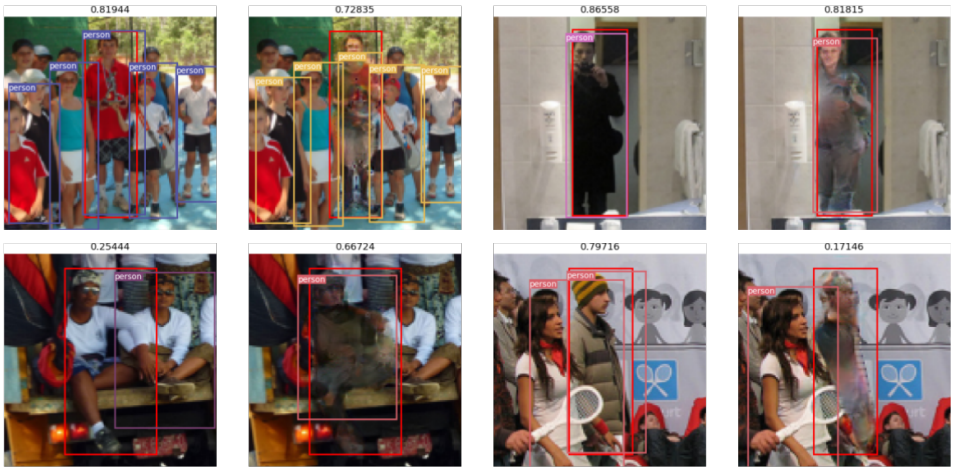


Figure 7.11: Comparison of detections on both original (first and third columns) and generated (second and fourth columns) person images. You can see a value of IoU above every image. The ground-truth bounding box is drawn in red.

the cost that some of the produced persons might not look well.

input	*mean IoU*
original	0.76535
generated	0.65729

Table 7.8: Mean IoU of detections given by YOLO v3 detector.

7.5 Augmenting the dataset with person generator

As a use-case of the proposed person generator, I show how it can be used to augment a dataset for person detection.

To evaluate the influence of such an augmentation, I train a classifier similar to the one used in 5.3 on a small dataset. I evaluate the performance of the classifier in 6 different experiments where each augments the dataset in a different way. The basic experiments are described in table 7.9 and the resulting 6 experiments are created by their combination. The shortcuts used in the table are the same that were used previously in table 5.1 and GAN denotes whether the person images were augmented by generator from GAN. This augmentation is done by picking one person and changing its appearance by person generator. Randomly picked examples of augmented images by person generator are shown in Figure 7.12.

For each experiment, I used a pretrained VGG network with changed

experiment name	*setting*						
	flip	br	contr	sat	hue	$\mathcal{N}(\mu, \sigma)$	GAN
baseline (base)	0	0	0	0	0	(0,0)	×
augmented (A)	0.5	0.1	0.1	0.1	0.1	(0,3)	×
GAN (G)	0	0	0	0	0	(0,0)	✓

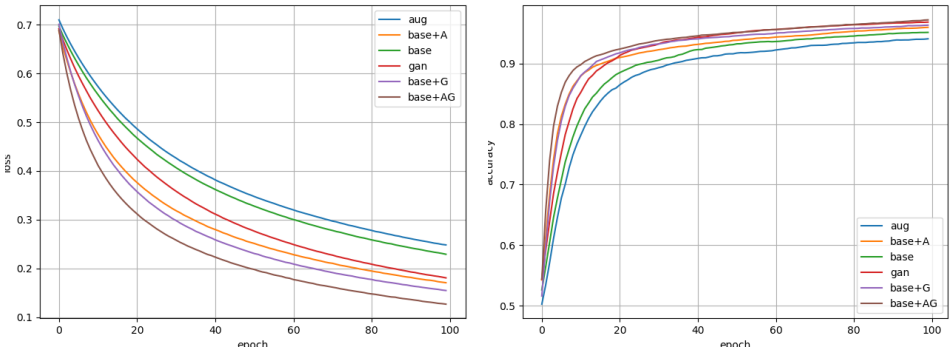
Table 7.9: Table containing names of performed experiments together with their setup.

Figure 7.12: Randomly taken examples of augmentation by person generator.

classification layer and trained it for 100 epochs, with Adam optimizer, binary cross entropy loss and learning rate of $5 \cdot 10^{-5}$. The training and testing process is repeated 10 times, and the results are averaged to suppress the stochasticity of the training". Curves for training are drawn in Figures 7.13, for validation in Figures 7.14 and average test results are shown in bar plots in Figures 7.15.

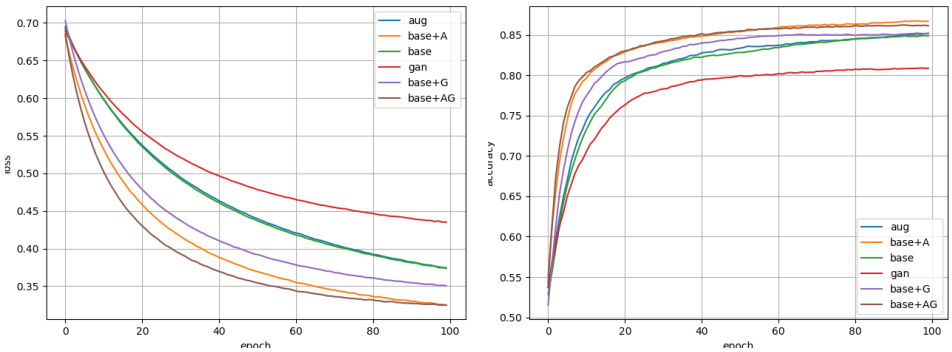
The results show that the use of dataset augmentation with person generator improves the loss and accuracy on the test set when compared to the setups without it. In Figure 7.14, where loss and accuracy validation curves are shown, is clearly visible that training with the use of this augmentation technique speeds up the training mostly in the earlier stages of the training despite the fact that the produced results are not perfect. However, for the purposes of use of such a network in a pedestrian detection pipeline in autonomous cars, every increase (no matter how small) of the performance counts since it might make the difference between detecting and not detecting a pedestrian in the end.

From the results it follows that the dataset augmentation with GANs can lead to improved performance of the classifiers trained on such datasets.



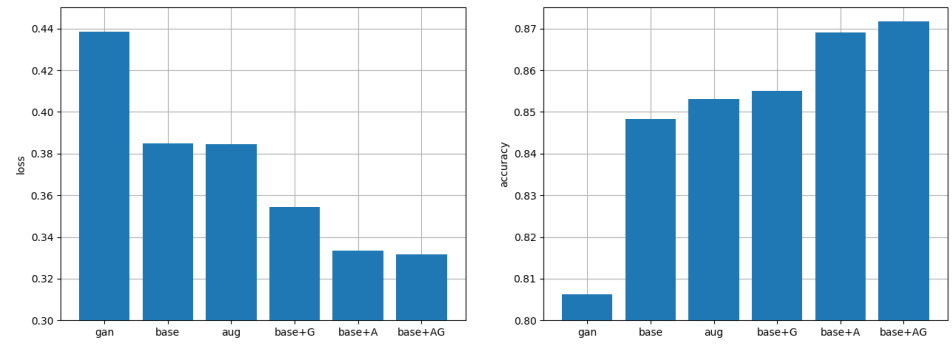
(a) : Loss. **(b) :** Accuracy.

Figure 7.13: Loss and accuracy on the training set.



(a) : Loss. **(b) :** Accuracy.

Figure 7.14: Loss and accuracy on the validation set.



(a) : Loss. **(b) :** Accuracy.

Figure 7.15: Loss and accuracy on the testing set with respective standard deviation.

Chapter 8

Conclusions

In the first part of this thesis, I described the theoretical background behind the methods used in this thesis. I started with the general overview of machine learning 2.1 followed by a description of artificial neural networks (section 2.2) and finally provided a more in-depth description of Generative Adversarial Networks in section 2.4.

In the second part, I explained the implementation of proposed methods and corresponding experiments. First, in chapter 4.2, I described the newly created datasets for the purposes of this thesis. There are two of these datasets. One is for the classification of image patches into one of the classes $\{\text{contains person}, \text{doesn't contain person}\}$ and the second one serves as a training set for the proposed person image generator and contains rich additional information such as the location of person keypoints.

In chapter 5, I experimented with various classical image augmentation techniques. I even proposed to augment only the parts of the images containing the target instances (humans in this setup) in order to increase the variance of the samples in the dataset even more. This method is not commonly used in practice. I showed that using it can lead to even better results. I experimented with training of CNN classifier with differently augmented datasets. For the given setup, I found that a simple horizontal flipping resulted in the most significant improvement of the accuracy in the trained classifier and the addition of noise to the person instances only led to the biggest decrease in the loss.

In the chapter 6, I described the proposed person image generator based on the idea of GANs and their progressive growing. I explained traditional losses that are used to train such networks and proposed a novel loss called *edge loss* that enforces the identity preservation in the domain of local image gradients instead of in the color domain. Later in the chapter, I gave an overview of the training procedure used to train the network.

In chapter 7, I performed experiments with the person image generator. First, I described the setup of the experiments. In the next section, I evaluated

the samples generated with two of the proposed methods and found out that the setup with the SPADE generator yielded better result most probably due to the better-used normalization that uses more of the input information. The proposed framework has novel ideas in its architecture. For example, a mask estimation network is used. I also performed an analysis of failure cases. To obtain a more elaborate evaluation of the quality of the generated samples, I did two surveys and obtained more than one hundred responses in each of them. From these surveys, it follows that the generated images are quite good, but humans are still able to distinguish between the original and generated image without any big problems. Further evaluation is done with the use of a human detector. I compared its performance on the generated images to the performance on the real ones. The average intersection over union of the generated person was overall smaller. However, the difference was not that dramatic (0.657 on generated images vs. 0.763 on the real ones). On some of the generated samples, the detector performed even better than on their original counterparts. In the last section of this chapter, I performed an experiment with augmenting a dataset for person detection with the use of this person generator. I showed that the performance of the classifier improves when it is trained on the dataset augmented with person generator despite the fact that the overall quality of the generated samples might not be that good. This leads to the conclusion that the increased quality of the generated samples might possibly lead to even better increase in the classifier performance in the future.

To sum up, I can state that all the steps that were required to be done in this thesis were successfully completed.

1. I researched existing data augmentation methods for neural networks and even proposed ones that are not commonly used.
2. I obtained two rich datasets for the training of the neural networks.
3. I proposed and implemented the individual components of a data augmentation system, proposed new ones, and performed various experiments evaluating the impact of these components.
4. I proposed and implemented a person generation network that can serve for image data augmentation. This network contains novel ideas such as mask estimation network and edge loss.
5. I chose a suitable neural network and trained it in the augmented dataset. I showed that the accuracy of the network trained on the dataset augmented with the proposed method increases.

Appendices

Appendix A

Bibliography

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[2] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[3] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[4] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Simultaneous detection and segmentation. In *European Conference on Computer Vision*, pages 297–312. Springer, 2014.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] Jan Drchal. Statistical machine learning, lecture 8: Deep neural networks, 2017.

[7] Adit Deshpande. A beginner's guide to understanding convolutional neural networks part 2, 2016. URL `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/`.

[8] Bob Fergus. Neural networks, mlss 2015 summer school, 2015. URL `http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf`.

[9] Jan Drchal. Statistical machine learning, lecture 5: Artificial neural networks, 2017.

[10] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[11] Faisal Shahbaz. Five powerful cnn architectures, 2018. URL https://medium.com/datadriveninvestor/five-powerful-cnn-architectures-b939c9ddd57b.

[12] Siddharth Das. Cnn architectures: Lenet, alexnet, vgg, googlenet, resnet and more, 2017. URL https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[16] Taylor Mordan, Nicolas Thome, Gilles Henaff, and Matthieu Cord. Revisiting multi-task learning with rock: a deep residual auxiliary block for visual detection. In *Advances in Neural Information Processing Systems*, pages 1310–1322, 2018.

[17] Thalles Silva. A short introduction to generative adversarial networks, 2017. URL https://sthalles.github.io/intro-to-gans/.

[18] Lilian Weng. From gan to wgan, 2017. URL https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html.

[19] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[20] Jon Gauthier. Conditional generative adversarial nets for convolutional face generation.

[21] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[22] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5767–5777, 2017.

[23] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Generative image inpainting with contextual attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5505–5514, 2018.

[24] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[25] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[26] Abdenour Hadid. The local binary pattern approach and its applications to face analysis. In *2008 First Workshops on Image Processing Theory, Tools and Applications*, pages 1–9. IEEE, 2008.

[27] Sneha H.L. Pixel intensity histogram characteristics: Basics of image processing and machine vision, 2017. URL `https://www.allaboutcircuits.com/technical-articles/image-histogram-characteristics-machine-learning-image-processing/`.

[28] Wikipedia. HSL and HSV — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=HSL%20and%20HSV&oldid=890683083`, 2019. [Online; accessed 30-April-2019].

[29] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Free-form image inpainting with gated convolution. *arXiv preprint arXiv:1806.03589*, 2018.

[30] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*, 2018.

[31] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8798–8807, 2018.

[32] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. *arXiv preprint arXiv:1903.07291*, 2019.

[33] Liqian Ma, Xu Jia, Qianru Sun, Bernt Schiele, Tinne Tuytelaars, and Luc Van Gool. Pose guided person image generation. In *Advances in Neural Information Processing Systems*, pages 406–416, 2017.

[34] Liqian Ma, Qianru Sun, Stamatios Georgoulis, Luc Van Gool, Bernt Schiele, and Mario Fritz. Disentangled person image generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 99–108, 2018.

[35] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[36] Tom M Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2. URL `http://www.worldcat.org/oclc/61321007`.

[37] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT Press, 2012.

[38] Allen Huang and Raymond Wu. Deep learning for music. *arXiv preprint arXiv:1606.04930*, 2016.

[39] Arthur L Samuel. Some studies in machine learning using the game of checkers. ii—recent progress. In *Computer Games I*, pages 366–400. Springer, 1988.

[40] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[41] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[42] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[43] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[44] Volodymyr Mnih and Geoffrey E Hinton. Learning to label aerial images from noisy data. In *Proceedings of the 29th International conference on machine learning (ICML-12)*, pages 567–574, 2012.

[45] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[46] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[47] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[48] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

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

[50] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[51] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[52] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405 (6789):947, 2000.

[53] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[54] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[55] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[56] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL `http://ruder.io/optimizing-gradient-descent/index.html`.

[57] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[58] Jason Brownlee. Gentle introduction to the adam optimization algorithm for deep learning, 2017. URL `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`.

[59] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

[60] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

[61] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[62] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[63] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[65] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[66] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[67] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[68] Martin J Osborne and Ariel Rubinstein. *A course in game theory*. 1994.

[69] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

[70] Martin Arjovsky and Leon Bottou. Towards principled methods for training generative adversarial networks. 2017.

[71] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[72] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[73] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[74] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

[75] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. In *ACM Transactions on Graphics (ToG)*, volume 28, page 24. ACM, 2009.

[76] James Hays and Alexei A Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics (TOG)*, 26(3):4, 2007.

[77] Alexei A Efros and William T Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346. ACM, 2001.

[78] Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1033–1038. IEEE, 1999.

[79] Rolf Köhler, Christian Schuler, Bernhard Schölkopf, and Stefan Harmeling. Mask-specific inpainting with deep neural networks. In *German Conference on Pattern Recognition*, pages 523–534. Springer, 2014.

[80] Li Xu, Jimmy SJ Ren, Ce Liu, and Jiaya Jia. Deep convolutional neural network for image deconvolution. In *Advances in Neural Information Processing Systems*, pages 1790–1798, 2014.

[81] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Globally and locally consistent image completion. *ACM Transactions on Graphics (ToG)*, 36(4):107, 2017.

[82] Timo Ojala, Matti Pietikainen, and David Harwood. Performance evaluation of texture measures with classification based on kullback discrimination of distributions. In *Proceedings of 12th International*

Conference on Pattern Recognition, volume 1, pages 582–585. IEEE, 1994.

[83] T. Ojala, M. Pietikainen, and T. Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24 (7):971–987, July 2002. ISSN 0162-8828. doi: 10.1109/TPAMI.2002.1017623.

[84] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pages 59–66. IEEE, 1998.

[85] Computer Graphics staff. Status report of the graphic standards planning committee. *SIGGRAPH Comput. Graph.*, 13(3):1–10, August 1979. ISSN 0097-8930. doi: 10.1145/988497.988498. URL `http://doi.acm.org/10.1145/988497.988498`.

[86] Denis Simakov, Yaron Caspi, Eli Shechtman, and Michal Irani. Summarizing visual data using bidirectional similarity. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[87] Soheil Darabi, Eli Shechtman, Connelly Barnes, Dan B Goldman, and Pradeep Sen. Image melding: Combining inconsistent images using patch-based synthesis.

[88] Kaiming He and Jian Sun. Image completion approaches using the statistics of similar patches. *IEEE transactions on pattern analysis and machine intelligence*, 36(12):2423–2435, 2014.

[89] Yijun Li, Sifei Liu, Jimei Yang, and Ming-Hsuan Yang. Generative face completion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3911–3919, 2017.

[90] Yinan Zhao, Brian Price, Scott Cohen, and Danna Gurari. Guided image inpainting: Replacing an image region by pulling content from another image. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1514–1523. IEEE, 2019.

[91] Saining Xie and Zhuowen Tu. Holistically-nested edge detection. In *Proceedings of the IEEE international conference on computer vision*, pages 1395–1403, 2015.

[92] Kilian Q Weinberger and Lawrence K Saul. Unsupervised learning of image manifolds by semidefinite programming. *International journal of computer vision*, 70(1):77–90, 2006.

[93] Lawrence K Saul and Sam T Roweis. Think globally, fit locally: unsupervised learning of low dimensional manifolds. *Journal of machine learning research*, 4(Jun):119–155, 2003.

[94] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[95] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[96] Paul Viola, Michael Jones, et al. Rapid object detection using a boosted cascade of simple features.

[97] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[98] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *international Conference on computer vision & Pattern Recognition (CVPR'05)*, volume 1, pages 886–893. IEEE Computer Society, 2005.

[99] Guha Balakrishnan, Amy Zhao, Adrian V. Dalca, Fredo Durand, and John Guttag. Synthesizing images of humans in unseen poses. In *CVPR*, 2018.

[100] Haoye Dong, Xiaodan Liang, Ke Gong, Hanjiang Lai, Jia Zhu, and Jian Yin. Soft-gated warping-gan for pose-guided person image synthesis. In *NeurIPS*, 2018.

[101] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[102] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

[103] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[104] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, Sep. 2010. ISSN 0162-8828. doi: 10.1109/TPAMI.2009.167.

[105] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.