Chair of Computational Modeling and Simulation
TUM School of Engineering and Design
Technical University of Munich

Deep Learning based integration of manual changes on floor plans

Scientific work to obtain the degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design of the Technical University of Munich.

Supervised by Prof. Dr.-Ing. André Borrmann
Jimmy Abualdenien
Sebastian Esser
Chair of Computational Modeling and Simulation

Submitted by Janina Weidinger

e-Mail:

Submitted on 30. September 2022

Abstract

The uprising of Building Information Modeling (BIM) and the accompanying digitization of construction processes entail a substantial shift in established workflows. In this context the simultaneous existence of printed floor plans and the corresponding digital BIM model poses a major problem to consistency in planning processes. This coexistence necessitates the concurrent analog to digital translation of handwritten modifications, which is currently executed manually in a time intensive and error prone manner in practice. In this thesis a pipeline for the automated transfer of handwritten annotations from floor plan scans into the according BIM model is proposed. Initially, the input floor plan gets translated into a valid digital representation, that incorporates the performed modifications. Subsequently, the matching excerpt within the BIM model is extracted, differences are detected and finally integrated into the BIM model. Aiming at a practical implementation of this pipeline, the automated translation of handwritten annotations into established machine readable format constitutes the critical missing component. Consequently, the main part of this thesis focuses on developing a suitable approach for realising this first step of translating pixel based, annotated floor plans into their valid digital representations. Related work proposes a deep learning based Generative Adversarial Network (GAN) for similar image to image translation tasks requiring a comparably small dataset for good prediction results. In combination with a suitable hyperparameter optimization based on educated guessing, this GAN architecture is shown to be applicable for the given task. For training and testing the prototype model a dataset comprising 350 unique real world floor plan samples is created by simulating handwritten annotations manually using a digital pen. Visual inspection of the results shows a proof of concept for the desired automated translation of manual changes within pixel-based floor plan images. A high translation accuracy is obtained for (1) symbols with a significant representation in the dataset, (2) symbols that represent "add" or "delete" operations and (3) symbols with spatial isolation to other feature elements and annotations. An extensive and diverse dataset, adequate post processing and a suitable embedding of the presented approach into prevalent construction software applications are further steps for establishing a commercial, deep learning based routine for the automated integration of manual changes in printed floor plans into the corresponding digital BIM model.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

2D	two-dimensional
3D	three-dimensional
AI	Artificial Intelligence
ANN	Artificial Neural Network
BIM	Building Information Modeling
cGAN	conditional Generative Adversarial Network
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
FP	Floor Plan
GAN	Generative Adversarial Network
GD	Gradient Descent
GPU	Graphics Processing Unit
GUI	graphical user interface
IFC	Industry Foundation Classes
LOD	Level Of Detail
LOG	Level Of Geometry
LOI	Level Of Information
ML	Machine Learning
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RIT	Rakuten Institute of Technology
SSE	summed squared error
TPU	Tensor Processing Unit

Chapter 1

Introduction

In the following, the motivation of deep learning in context of translating manual changes on floor plans into digital symbols is illustrated. Further, the structure of the thesis is explained.

1.1 Motivation and idea of the thesis

Nowadays Building Information Modeling (BIM) gets more and more important in the wide field of the construction industry. BIM simplifies many steps of a building's lifecycle, makes work more efficient and is not prone to human errors. Also the interoperability between different tools is a big advantage of BIM.

However, the transition to use BIM in every field of application requires a big restructuring and is therefore a huge challenge for many engineering offices. Original working practices need a reshuffling, the staff has to learn new computing skills and different arrangements in salary and contracts are necessary between the project members and the client.

Furthermore, drawings are more practical on construction sites and can provide a high level of detail, which is difficult and expensive to model in three-dimensional (3D). Particularly for small companies with a lack of digital resources the implementation of BIM is a big obstacle MIGILINSKAS et al., 2013. A solution is needed, that enables companies to still obtain original working practices using printed floor plans and drawings, but also updates currently the digital model. The consistency of both the semantic and geometric part should be kept.

Artificial Intelligence (AI) provides big opportunities in automating construction processes, in consistency checking, in clash detection, in translation tasks and in various other applications. There are more an more established processes including various AI assistance in the wide field of BIM.

Thus, this thesis intends to combine the power of AI and the need for a coexistence of drawings and digital BIM models in order to find a solution for the first step of automatically translating handwritten modifications on a drawing into its corresponding digital BIM model. The goal of this thesis is to create a prototype deep learning based model, that automatically detects handwritten annotations in floor plans, translates them pixel wise into digital symbols and finally scales, rotates and places the symbols correctly within the floor plan image.

Aiming to get appropriate data for training this deep learning based model, an extensive research for online available floor plans is conducted within the scope of this thesis. On the basis of online available real world floor plans, a dataset is created particularly adjusted to the task of analog to digital symbol translation embedded into floor plan images.

Following the purpose of this thesis, a suitable architecture should be found by researching similar deep learning based image to image translation tasks. Further, the model should be trained using the adapted and best possible pre-processed dataset, optimised with respect to hyperparameters and finally evaluated based on visual inspection of the output.

The thesis also introduces a possible workflow for further integrating the translated digital symbols into the actual 3D BIM model. This tool should enable engineers to fully concentrate on their actual work and still having the possibility to contribute their work to the digital model without the necessity to learn many new digitization skills. Thus, the thesis will help to allow the coexistence of drawings and BIM models and further to provide consistency between both even with modifications on only one of them.

1.2 Structure of the thesis

In the first part of the thesis the fundamental concept of AI is explained giving a focus to neural networks and especially to both Convolutional Neural Network (CNN) and GAN model architectures.

The second part introduces related works, that also provide deep learning based solutions for facilitating the coexistence of analog drawings and the according BIM model by automating design, translation and linkage processes.

Chapter 4 outlines the concept of this thesis. The approach of using a GAN architecture for solving the task of this thesis is explained. Further, a pipeline for the full process of transferring annotations from analog floor plans into the corresponding BIM model is proposed.

The next chapter provides an extensive research and summary of online available floor plan datasets designed for similar applications and introduces the creation pipeline, important characteristics and some dataset statistics of the H-Symb FP dataset, which was established in context of this thesis.

Chapter 6 explains the pre-processing steps of the floor plan images, the implementation of the generator and discriminator architecture as well as their loss functions and also the training procedure. It further mentions some encountered problems during training.

The final part evaluates the performance of the model based on loss curves and visual inspection of the predicted images. Additionally, the impact of changing hyperparameters is discussed. The results are presented and some possible improvements regarding the dataset and model architecture are suggested. The thesis is concluded with a short summary of the results and some future ideas.

Chapter 2

Background - Fundamentals of AI

AI is a broad field of computer science, where artificial systems are created, that perform tasks requiring any type of human intelligence JAKHAR and KAUR, 2020. The final goal of AI is to bring computers to a level such that they are able to act and perform similar to humans when solving certain tasks like learning, vision, language and robotic motions GALLANT, 1993, p. 3.

The term intelligence plays a great role within the field of AI but is rather vague. Thus, there are various definition approaches for intelligence having their focus on different characteristics. In the field of AI according to Legg intelligence can be defined as "an agent being able to achieve goals in a wide range of environments" LEGG and HUTTER, 2007.

This means, that firstly a machine or artificial system is acting as a kind of agent. It secondly must be able to somehow interact with its environment by receiving and sending signals in other words by having an input and output possibility and by carrying out actions or calculations. And thirdly the agent needs an objective, which can be either known in advance or is unknown. By getting rewards from its environment the agent can classify its action, further determine its actual goal and tries to find a way to finally achieve this goal. In other words the key abilities of an intelligent artificial system are Learning, Acting and Thinking, illustrated in fig. 2.1.

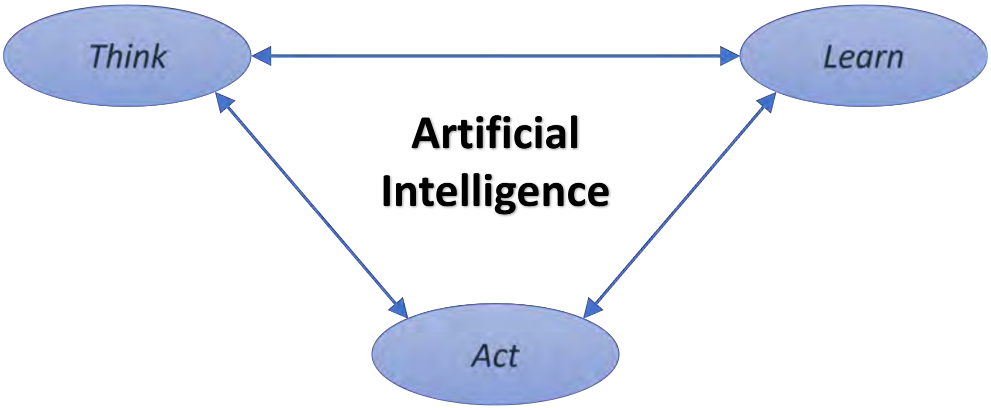


Figure 2.1: Key features of Artificial Intelligence

The branch Machine Learning (ML) of the generic term AI concentrates on the ability of a machine to learn dynamically from datasets without being explicitly programmed and thus modifies itself, when new data is available JAKHAR and KAUR, 2020. Section 2.1 gives an overview of ML. Deep Learning (DL) is in turn a subset of ML having its focus on imitating the architecture of the biological neural networks in brains within computational models and algorithms. This structure is called Artificial Neural Network (ANN) and is explained in more detail in section 2.2. The fig. 2.2 shows the umbrella term AI and how it comprises the contents of the fields ML and DL. GÉRON, 2019

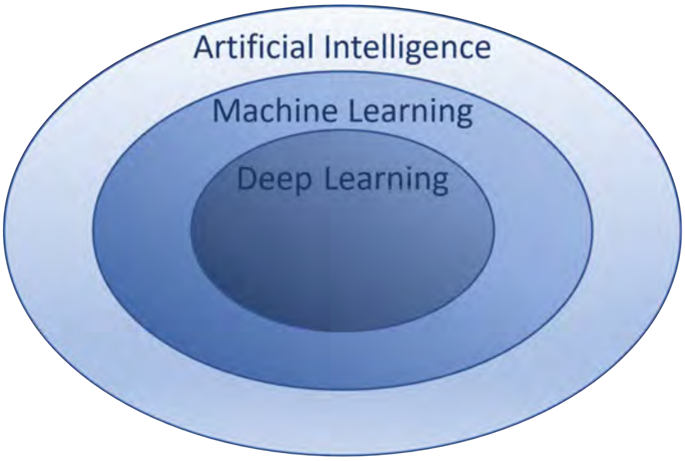


Figure 2.2: The dependencies of the umbrella term AI and its subsets ML and DL

2.1 Basic principle of Machine Learning

ML concentrates on writing code in order to give a computer the opportunity to learn from data GÉRON, 2019. Historically ML belongs to the field of control science also called cybernetics, because the original learning algorithms providing a convergence and convergence rate of the learning process came up within cybernetics. With the improving performance of computers ML is nowadays better known to belong to the area of computer science. FRADKOV, 2020

In the following sections the application and concept of ML are described. Further the most common categories of ML systems are explained depending on their way of learning. There are three main learning properties to classify a ML algorithm. These categories can be combined independently. The amount of supervision, the ability to learn stepwise and the way of generalizing the target value are these three characteristics for classifying a learning algorithm.

2.1.1 Applications and Concept of Machine Learning

ML is useful for problems, that are either too complex for any previous approaches or where no traditional algorithm can be applied. Additionally code containing many rules can often be simplified, shortened and improved in terms of performance by ML algorithms. When dealing with constantly changing or new data ML can be able to adapt the algorithms constantly to the current environment. It is further common to use ML for assisting humans to understand large or complicated datasets better by finding e.g. concealed correlations or other hidden patterns. This area of application is called data mining. GÉRON, 2019

The big difference between traditional programming and ML is, that in traditional programming there already exist both data as well as a program running on a computer and producing an output. In ML data and the desired output exist and the computer produces a program, that can be used for the evaluation of further unknown data to produce an output. This difference is illustrated in fig. 2.3. BROWNLEE, 2015

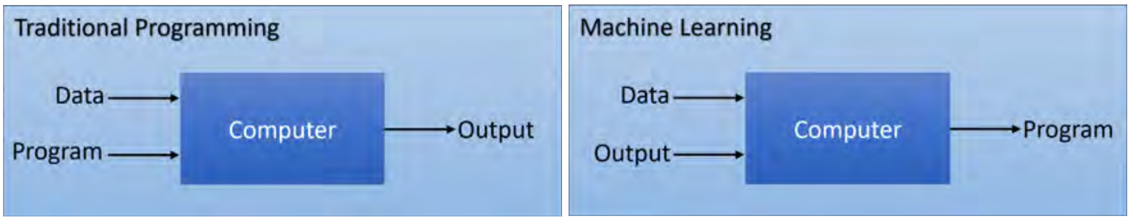


Figure 2.3: Concept of traditional programming compared with ML BROWNLEE, 2015

The typical workflow to create a ML system is described in fig. 2.4. Firstly the data needs to be collected. As a second step, the dataset is cleaned by deleting unnecessary features as well as invalid data instances. Important features are selected according to their influence on the target variable and if needed the data is manipulated. Then, the dataset is split into a training and a test set. Thirdly the algorithm is selected and the model is trained using the training dataset. To evaluate the model, the test set is used. As last step the algorithm and feature selection are improved and the model is trained and tested again. BROWNLEE, 2015; GÉRON, 2019; PANT, 2019

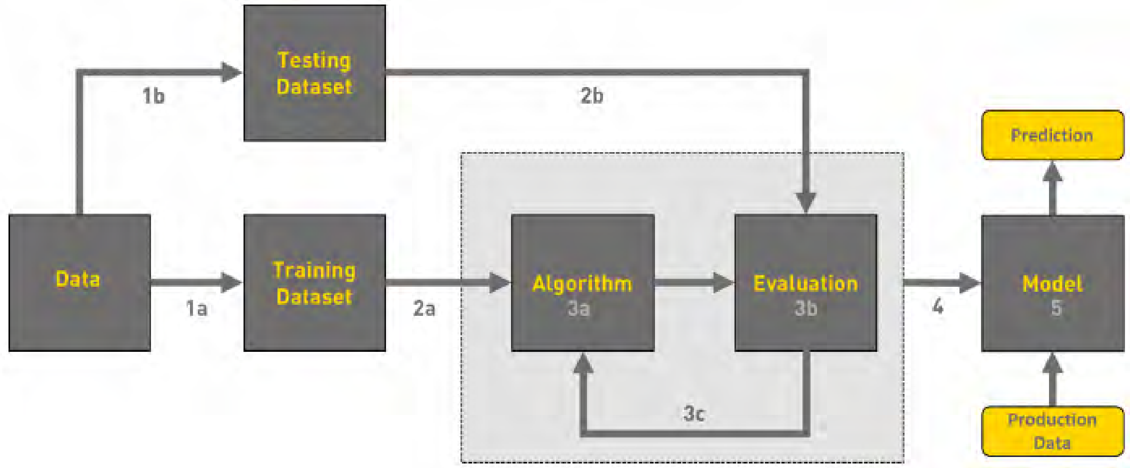


Figure 2.4: Typical workflow of developing a ML system PANT, 2019

2.1.2 Types of Supervision

The amount and the type of human supervision during the learning phase for the ML system is an important category. It states how the dataset needs to be preprocessed in order to train the system. In fig. 2.5 the different types of supervision are visualized including their most common application problems. In the following section the different types of supervision and some common applications for each type are described.

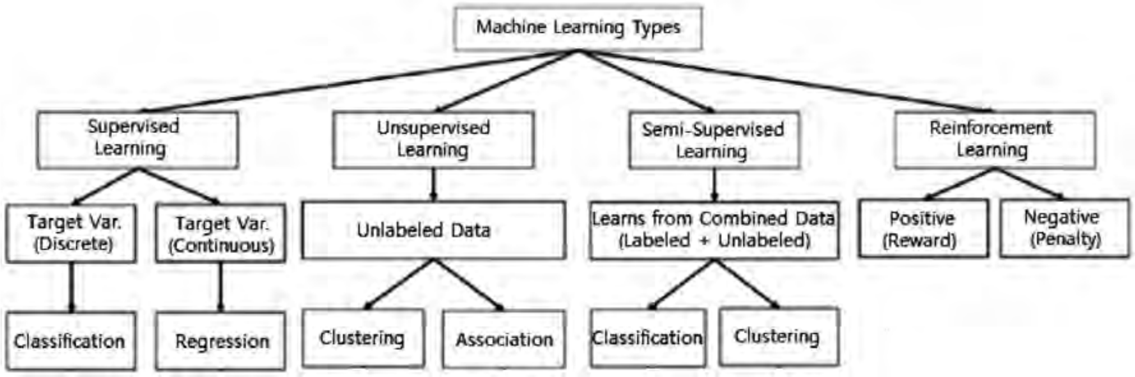


Figure 2.5: Different types of Supervision during training of ML systems with properties SARKER, 2021

Supervised Learning During supervised learning a function learns how to map a certain input to an output. This function is trained based on an input dataset with a labelled output for each input sample. Thus, the training dataset needs to be prepared in advance including a fitting output for each sample. GÉRON, 2019

Typical tasks for supervised learning are regression and classification. When the output parameter - also called the target variable - is continuous, the data is fit into a function, that identifies the fitting output. This is called regression. When the target variable is discrete, the problem is called classification. The data is classified by a function, that separates the input data into discrete categories and returns the mapped category according to the input sample. Supervised learning is applied, when the goal is known and it is possible to achieve this goal using a certain input set. This is called a task-driven approach. GÉRON, 2019; SARKER, 2021; TAMBE, 2020

Unsupervised Learning A ML system learns unsupervised, when the dataset is without any labelling, thus it learns without any human interference. Unsupervised learning often helps to get a better understanding of a dataset by applying certain strategies of data mining to the dataset. There are different algorithms, that detect certain properties, sort or restructure the dataset. GÉRON, 2019

A common application of unsupervised learning is the clustering of the dataset. Some algorithms like K-Means or the Hierarchical Cluster Analysis (HCA) sort the dataset into different subgroups with similar properties. It is also possible to detect anomalies or find novelties, being instances, that have rather different or new features compared to the rest of the dataset. Another important application of unsupervised learning is the visualization and dimensionality reduction. When applying algorithms to the dataset like for instance the Principal Component Analysis (PCA) or the t-Distributed Stochastic Neighbor Embedding (t-SNE) the data can be visualized keeping as much structure of the dataset as possible in order to understand hidden patterns or relationships better and also the dimensionality of a dataset can be reduced. This means, that the dataset is simplified by merging certain relating features into one feature. Dimensionality reduction can decrease the computation time and storage space for further data processing a lot. It is also possible to do association rules mining with a dataset in order to detect relationships and dependencies in large datasets. Cios et al., 2007; Géron, 2019; Sarker, 2021; Tambe, 2020

Semisupervised Learning When a dataset is labelled only partially it is called semisupervised learning. It is a kind of hybridization as most semisupervised learning algorithms are combinations of both supervised and unsupervised algorithms. The system learns using both labelled and unlabelled data. Géron, 2019

A common semisupervised algorithm is the deep belief network (DBN) consisting of unsupervised components, that are finally put together using supervised algorithms. A famous application for semisupervised learning is for example the automatic recognition of people in different pictures. The user is asked by the application to name a detected person only once producing a small amount of labelled data and the algorithm identifies automatically other pictures with this person and labels this person by itself. Géron, 2019; Sarker, 2021; Tambe, 2020

Reinforcement Learning This kind of learning takes usually place within a particular context or an environment, where an agent learns by receiving penalties or rewards for certain actions. The agent is able to observe the environment and interact with it by selecting and performing possible actions and accordingly receiving rewards. Future actions are improved by maximizing the reward, calculated from previous action rewards. The development of the optimal action strategy to achieve the best reward happens automatically and is called policy. The aim of reinforcement learning is to improve the policy with each new similar problem and thus train the agent to act better when solving similar problems. Géron, 2019; Lorenz, 2020; Sarker, 2021

Reinforcement learning is an environment-driven approach. Typical applications of reinforcement learning are for example the training of robots to increase automation or to learn a winning policy for certain games like for instance the traditional chinese board game "Go". Géron, 2019; Lorenz, 2020; Sarker, 2021; Tambe, 2020

2.1.3 Incremental Learning

Another important characteristic of a ML system is whether it can learn stepwise and adapt to new data or whether the learning process is terminated and cannot benefit from new data without a new creation of the whole model. In fig. 2.6 the different workflows of batch and online ML are shown.

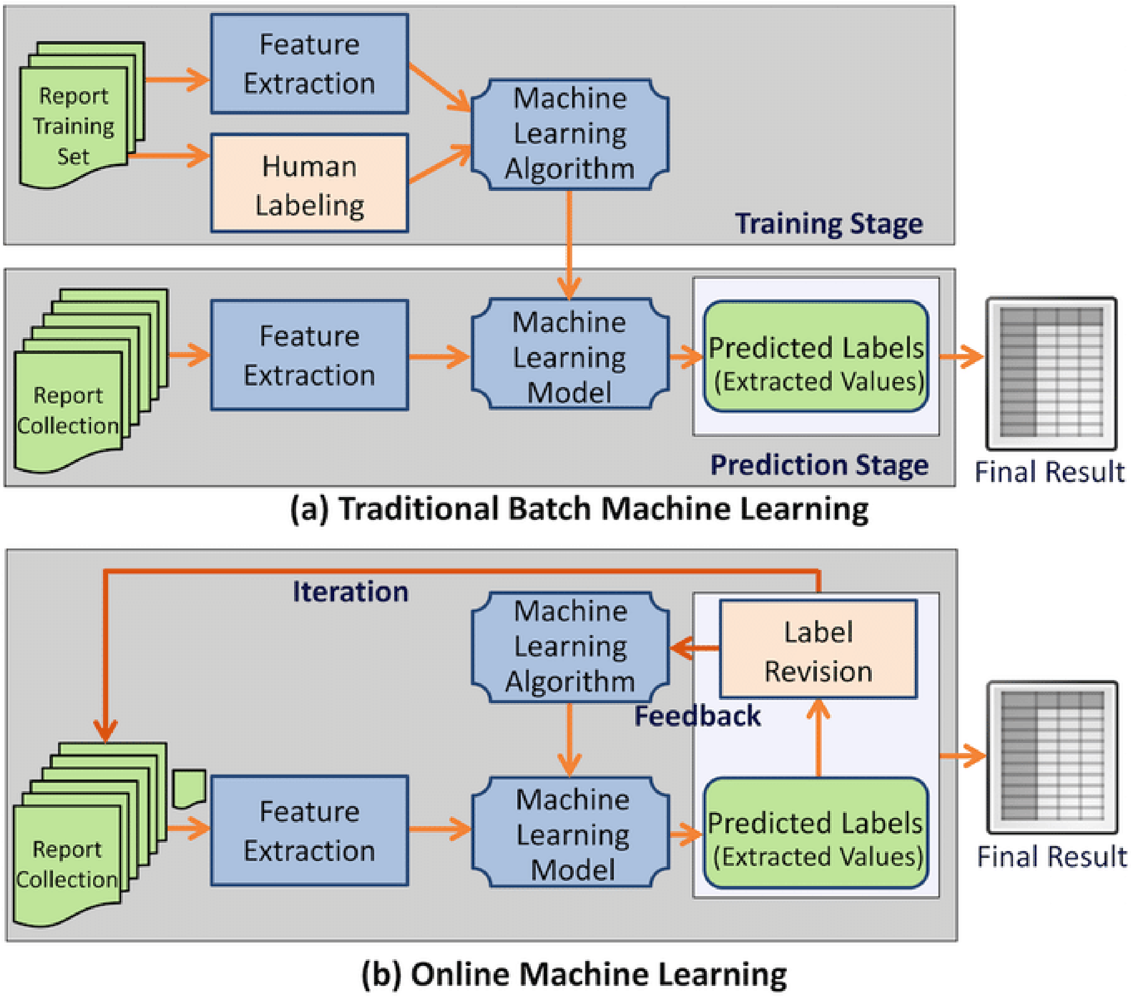


Figure 2.6: The workflow of batch versus online ML systems ZHENG et al., 2017

Batch Learning When a ML system uses batch learning, it needs to have all data available directly in the beginning. It is trained once using the dataset to create the ML model. Then, the model runs without any more learning. When new data is accessible, the whole model has to be trained again using both the old and new data. It thus might need a lot of computation time and storage space. Batch learning is also called offline learning. GÉRON, 2019; ZHENG et al., 2017

Online Learning An online ML system is updated stepwise and can also use new available data even if this data arrives as a continuous flow. The system is sequentially trained either on pieces of the dataset, called mini-batches or by single instances which are fed step by step to the algorithm. Accordingly, the system never stops learning. GÉRON, 2019; ZHENG et al., 2017

The computation of one learning iteration is easy and normally not time consuming. As long as there is no demand for bringing the system back into a previous state, it needs only the current mini-batch data available, which can save a lot of storage space. The learning rate of a ML system states, how fast the model adapts to changing data instances. A high rate effects the system fast and a low rate slower, but keeps the system more stable when outliers or other manipulating instances are fed to the learning algorithm. GÉRON, 2019; ZHENG et al., 2017

2.1.4 Type of Generalization

The task of a ML system is usually to predict an output based on certain input data. This output is generated depending on the way of abstracting the dataset and thus upon which generalization basis the prediction model is built. There are two different ways of generalizing the dataset.

Instance-Based Learning A ML system learns Instance-Based, when it predicts an output using a similarity measurement between the input data and previous data sample instances. The target variable is determined by calculating the highest similarity between the attributes of all training instances and the attributes of the input instance. Thus, the main task of Instance-Based Learning is to store the training data instances as effectively as possible to reduce storage and computation time for the similarity algorithm. The nearest neighbour method is one of the most common instance based learning algorithms. GÉRON, 2019; TAMBE, 2020

Model-Based Learning In Model-Based Learning training data is used to build a ML model. This model can then further be used for predictions, when having the same input attributes as the model was trained with. After training, the model does not use single training instances any more. It calculates the output based on certain rules, that the model learned during training out of the training dataset. Model-Based Learning needs evaluation and mostly some model refinement in order to generate the best predicting model. BUCKLEY, 2012; GÉRON, 2019; TAMBE, 2020

2.2 Basics about Neural Networks

ANNs are inspired by the architecture of brains GÉRON, 2019. Nowadays they play a great role in solving large and complex problems and even outperform other Machine Learning techniques in some areas of application. The first modern ML application was called "perceptron" and could recognize the letters of the alphabet. It was developed by a group lead by the psychologist Frank Rosenblatt around 1950. It became the prototype of ANNs. FRADKOV, 2020

In the following section the basic structure of ANNs is explained. Further, training and learning strategies as well as optimization techniques are discussed.

2.2.1 Standard structure

The standard structure of an ANN as illustrated in fig. 2.7 consists of different layers, that are connected with each other. Within each layer there is a certain amount of nodes, also called neurons. Each connection between the nodes of two different layer has its own weight and each neuron has a bias. In this chapter each component of an ANN is described.

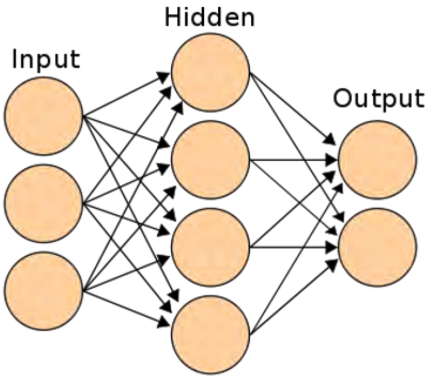


Figure 2.7: The basic structure of a shallow ANN, SRIVASTAVA, 2014

Input layer The first layer of an ANN is the input layer. Here the available data is passed into the network without any computation taking place, yet. The amount of input nodes depends on the problem and the amount of available input features. GÉRON, 2019

Hidden Layers The second part are the hidden layers. It is called 'hidden', because in contrast to the input and output layers, the computation taking place within this section is not visible to the outside world. The hidden layers can have many layers with different amount of nodes. The more hidden layers are present in the network, the deeper is the network.AGGARWAL, 2018; GÉRON, 2019

Each hidden layer applies another transformation to the input data and thus being able to split the problem into smaller parts that can be used to correlate an input to its output. The deeper the network, the more complex tasks can be solved but also the more computation time and storage space is necessary. In the first hidden layers the network deals with rather primitive features getting more into detail with each additional layer. For a single hidden layer network it is for example hard to capture a non linear pattern in the data. A multiple hidden layer ANN like in fig. 2.8 is called a Deep Neural Network (DNN) and a single hidden layer network as shown in fig. 2.7 is a shallow Network. AGGARWAL, 2018; GÉRON, 2019; GUPTA, 2021; SRIVASTAVA, 2014

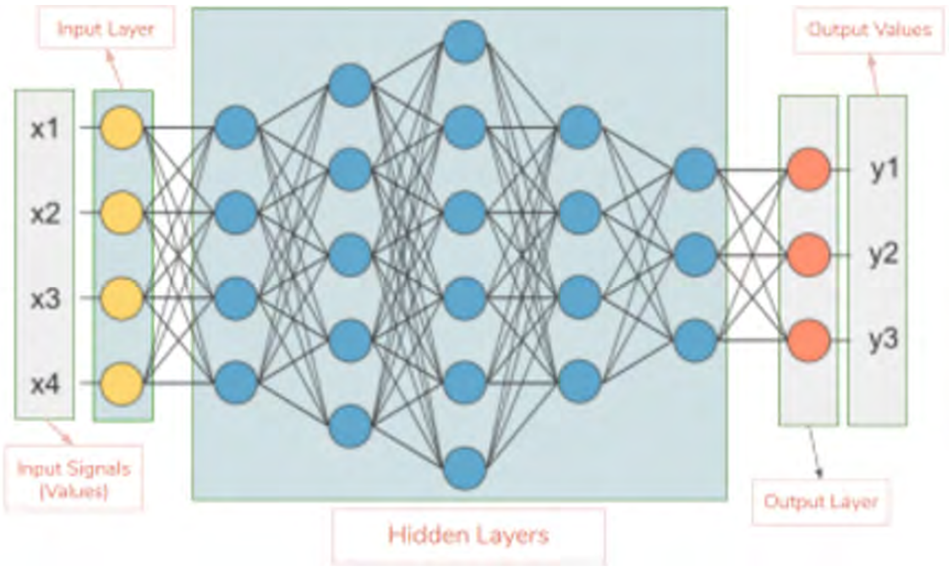
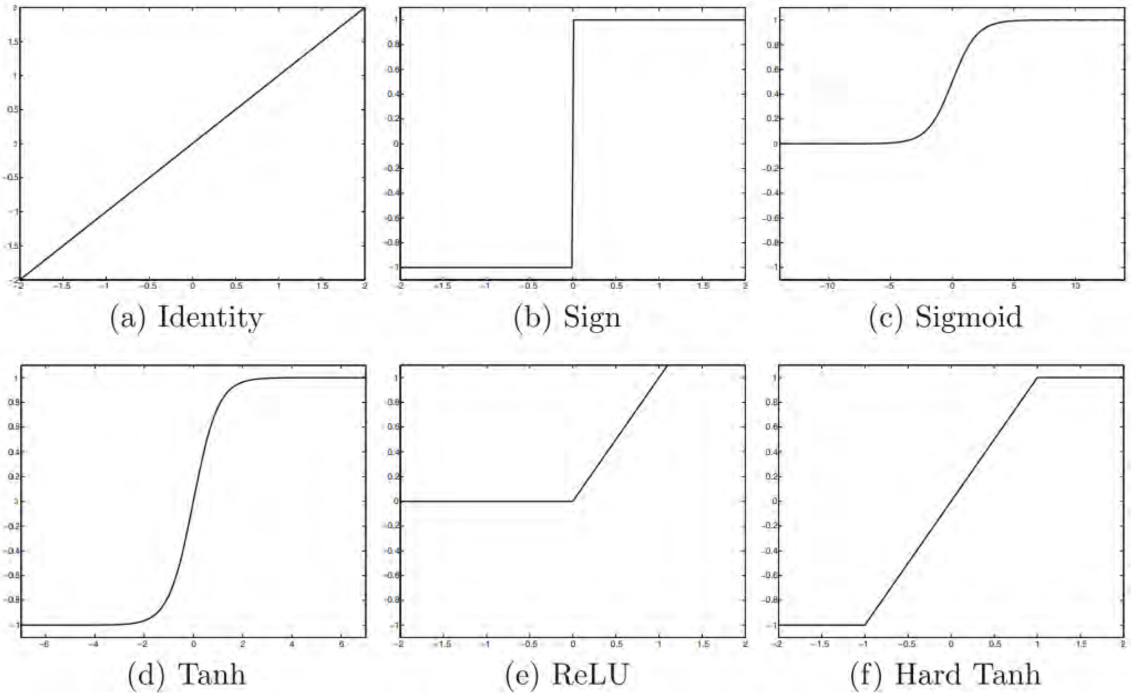


Figure 2.8: The architecture of a deep learning ANN with many hidden layers, GUPTA, 2021

Activation function The activation function is used on each node in the network to map a certain input value to an output usually between 1 and 0. This output value indicates the amount of activation of this node. The activation pattern of the last hidden layer in combination with the connection weights determines the output of the DNN. If a network uses only linear activation functions on each node, independent of the amount of hidden layers only linear behaviour can be observed within the network. A DNN using non-linear activation functions can further deal with non linear patterns. In fig. 2.9 some common non-linear activation functions are plotted. AGGARWAL, 2018; FENG et al., 2019; GÉRON, 2019; S. SHARMA, 2017


(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Figure 2.9: Common activation functions, AGGARWAL, 2018

Weights and Biases Each node of a layer is connected with each node of the following layer. This linkage has a certain weight. The weights determine the influence of a node from the previous layer on a node in the following layer. Thus, the weights show the importance of a node regarding the output. The higher a weight the more influence has this weight on the output. AGGARWAL, 2018; GÉRON, 2019

The bias is added to the value of each node to make up for the difference between the intended and the function's output. It must not necessarily be the same for each node. In the beginning all weights and biases are either assigned randomly or the values from a similar problem with an already trained network can be used to speed up the training process. During the training process the weights and biases are adapted. AGGARWAL, 2018; GÉRON, 2019

Output layer The last layer of an ANN is the output layer. It transfers the information obtained within the network to the outside. In opposite to the input layer it is possible to do computations within the output layer. AGGARWAL, 2018

2.2.2 Training with Backpropagation

Training of an ANN means, that the weights and biases are adapted in order to get an optimal mapping of the input to the correct output. Within the training process an existing dataset is used and for the performance evaluation of the network a test dataset is necessary including other samples than the training dataset. In shallow networks the weights and biases can be calculated rather easy, as the error of the output is a direct function of the weights and biases. In DNNs the error is a composition of the weights of many different layers making the computation complex. AGGARWAL, 2018

The Backpropagation algorithm is a supervised learning method to train feed-forward DNNs. Feed-forward means, that each node of a layer is connected with each node of the next layer and the connection goes only into the forward direction. Backpropagation normally applies the gradient descent method to achieve the optimal adaption of the weights and biases for each node in the hidden layers. Backpropagation consists of three steps called an epoch, that are repeated as many times as there exist training samples or until the error is acceptably small. One epoch consists of the following steps: AGGARWAL, 2018

1. *Forward Pass:* As a first step the model uses a sample z consisting of the input and output value pair (x, y) or a bunch of samples from the training dataset and calculates the predicted output. The output is determined by sending the data through the DNN using the current weights and biases. Opposite to a normal prediction of the model, the output of all intermediate neurons within the hidden layers are also stored for the backward pass. AGGARWAL, 2018; GÉRON, 2019

   The initial weights and biases before the first epoch are set to random values or to the values of a similar already trained problem in order to start closer to the final values. The initial weights should not be zero, as otherwise the Backpropagation algorithm might get stuck as its effect would be the same on all neurons with biases and weights set to zero. AGGARWAL, 2018; GÉRON, 2019; de WILDE, 1996

2. *Error Calculation:* Secondly, the error between the predicted output value $f(o_m)$ and the actual target value y_m is calculated for each neuron in the output layer. Here, o_m is the value, that is fed into the output activation function f. It is derived from the input value x from the training sample z consisting of the input output value pair (x, y). The output layer has M neurons. The sum of all errors is called the Error function or Loss function $\mathcal{L}(\theta, z)$. As soon as the function of one predicted output is nonlinear, the whole error function is also nonlinear. One of the most common error functions used for real-valued predictions is the summed squared error (SSE) function, which is defined in equation 2.1. AGGARWAL, 2018, p. 17; MUNRO, 2017; SEMENOV et al., 2019, p. 58; de WILDE, 1996

$$\mathcal{L}(\theta, z)_{SSE} = \sum_{m=1}^{M} (y_m - f(o_m))^2 \tag{2.1}$$

For discrete predictions usually the cross-entropy loss function is used in combination with the softmax activation function in the output layer. Softmax returns a vector with the sum of all elements being 1. Thus, Softmax normalizes all output values to a number between 0 and 1. Equation 2.2 shows the cross-entropy loss function for multidimensional ANNs. BHALLEY, 2021b, p. 210; AGGARWAL, 2018, p. 17; MUNRO, 2017; SEMENOV et al., 2019, p. 58 ff.

$$\mathcal{L}(\theta, z)_{CE} = -\sum_{m=1}^{M} y_m \log(f(o_m)) \tag{2.2}$$

3. *Backward Pass:* The goal of the backward pass is to minimize the error function of step two. To determine the weights any method for nonlinear minimization can be used. The most common method in ANNs is the Gradient Descent method explained in section 2.2.3. The weights are updated in decreasing order of the layers starting with the layer closest to the output layer and going backwards through the network. AGGARWAL, 2018

Gradient descent returns the optimal direction within the weight space for getting the maximum decrease of the error function, when the weight change is infinitesimal small. The step size towards the local minimum with which the weights are adapted, is called the learning rate η and determines the convergence velocity. Each weight w_j is accordingly updated proportional to the gradient of $\mathcal{L}(\theta, z)$ with a stepsize multiplied by η according to equation 2.3. GALLANT, 1993, p. 19-21; PAPER, 2018; RUDER, 2016; de WILDE, 1996

$$\widehat{w}_j = w_j - \eta \frac{\partial \mathcal{L}(\theta, z)}{\partial w_j} \tag{2.3}$$

The full schematic structure for all steps of the Backpropagation learning algorithm is summarized and visualized in figure 2.10.

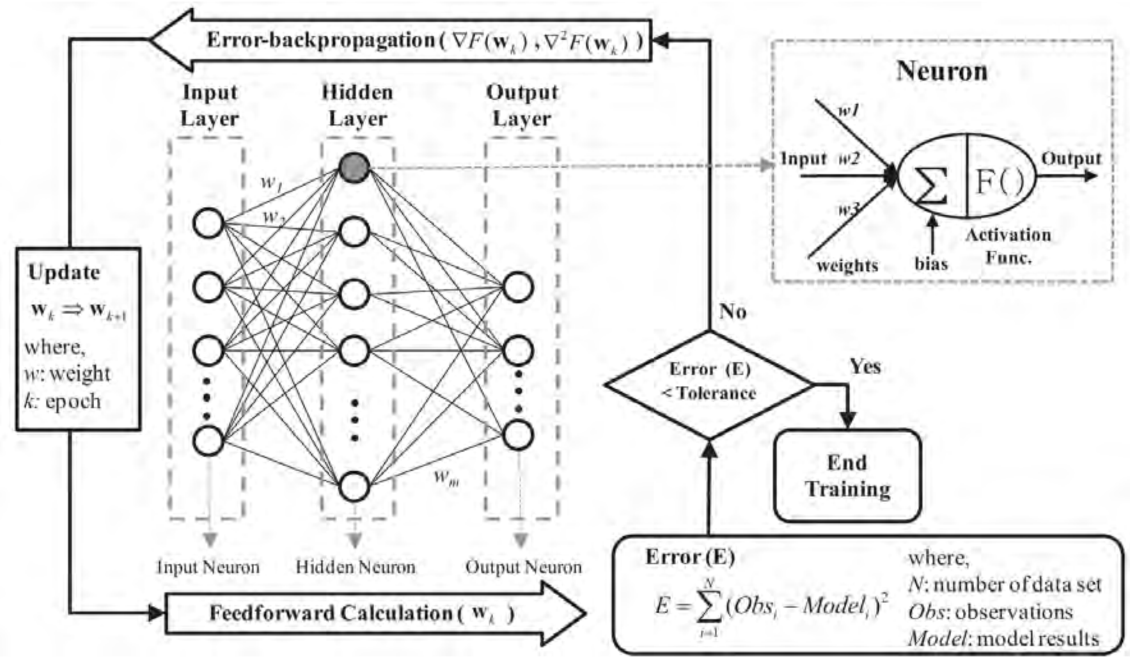


Figure 2.10: Structural Diagram of the Backpropagation training algorithm in a classical ANN, Kim and Seo, 2015

2.2.3 Gradient Descent

The Gradient Descent (GD) or also called steepest descent is the most common method to optimize neural networks. It can be applied on any differentiable function. Basically, the algorithm changes iteratively every parameter of a function in order to converge to a minimum. The objective function $\mathcal{J}(\theta, z)$ is described by its set of parameters θ and the data samples z used for training consisting of the input output pairs (x, y). Géron, 2019; Ruder, 2016

In ANNs the parameter set θ consists of all weights and biases within the model. And the objective function $\mathcal{J}(\theta, z)$ is the Loss function $\mathcal{L}(\theta, z)$, being for instance the SSE function $\mathcal{L}(\theta, z)_{SSE}$ from equation 2.1. When used within one epoch of the Backpropagation algorithm, GD returns the direction for all weights, that would minimize the error function $\mathcal{L}(\theta, z)$ fastest, assuming every step size would be infinitesimal small Paper, 2018. For the GD algorithm the following three basic steps are important to consider. Géron, 2019, chap. 4; Paper, 2018, p. 97 ff; Ruder, 2016

1. *Objective Function:* As a first step, the objective function $\mathcal{J}(\theta, z)$, that should be minimized is calculated. If GD should be applied on more than one training sample z_i with $i \in \mathbb{N}\{1, ..., N\}$ in one iteration step, $\mathcal{L}(\theta, z)$ needs to be averaged according to equation 2.4. N accounts for the amount of training samples, that are used within

one GD iteration step. BOTTOU, 2012

$$\mathcal{J}(\theta, z) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\theta, z_i) \tag{2.4}$$

2. *Partial Derivative:* Secondly, for implementing the GD algorithm the partial derivative of the objective function $\mathcal{J}(\theta, z)$ for each model parameter θ_j with $j \in \mathbb{N}\{1, ..., W\}$ needs to be calculated. W describes the amount of model parameter and in context of ANNs the number of weights and biases to be updated. The partial derivative of the objective function for parameter θ_j is written as $\frac{\partial}{\partial \theta_j}\mathcal{J}(\theta, z)$. The partial derivative can be solved using the chain rule and other techniques for calculating the derivatives of functions. PAPER, 2018, p. 97 ff

3. *Parameter Update*: Finally the partial derivative is used in combination with the stepsize or learning rate η to update the according weight or bias θ_j of the network. GÉRON, 2019, chap. 4; AGGARWAL, 2018, p. 121 ff.

$$\widehat{\theta}_j = \theta_j - \eta\frac{\partial \mathcal{J}(\theta, z)}{\partial \theta_j} \tag{2.5}$$

   When using a sufficiently small learning rate, it is assumed, that this GD approach converges linearly to the minimum BOTTOU, 2012.

Training issues

A good and fast convergence to the global minimum is a very important point for an effective network training. Thus, the learning rate needs to be adapted carefully and also depending on the shape of the objective function an optimisation of the GD algorithm needs to be taken into consideration.

Influence of the Learning rate GD converges to the local minimum depending on the learning rate η. Figure 2.11 illustrates, how the convergence changes with varying stepsize of the learning rate in a convex function. If η is too small, GD needs many iteration steps until it reaches the minimum as shown in subfigure 1 and thus training takes very long, but can take more detailed information into account. When the learning rate is too large as in subfigure 2, it might happen, that there is no convergence at all and the minimum is overshot. In subfigure 3 the learning rate is proportional to the gradient. The lower the gradient, the smaller the stepsize. In case of a minimum, the gradient is zero and thus, the minimum of the function is reached. GÉRON, 2019, chap. 4

Influence of the objective function shape When the objective function is not convex, it might have many local minima and plateaus. The gradient of every minimum, maximum and plateau is zero. Thus, GD often gets stuck in a non convex function and is not able to reach the global minima. It behaves as shown in figure 2.12. GÉRON, 2019, chap. 4;

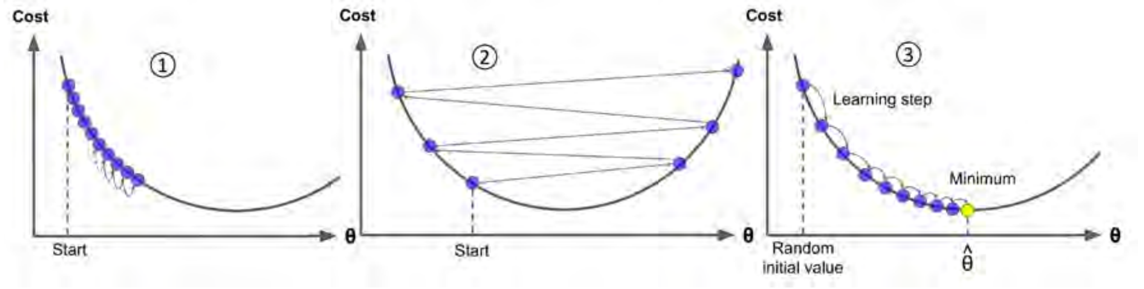


Figure 2.11: Convergence pattern of GD algorithm depending on the learning rate η in a convex function, GÉRON, 2019, chap. 4

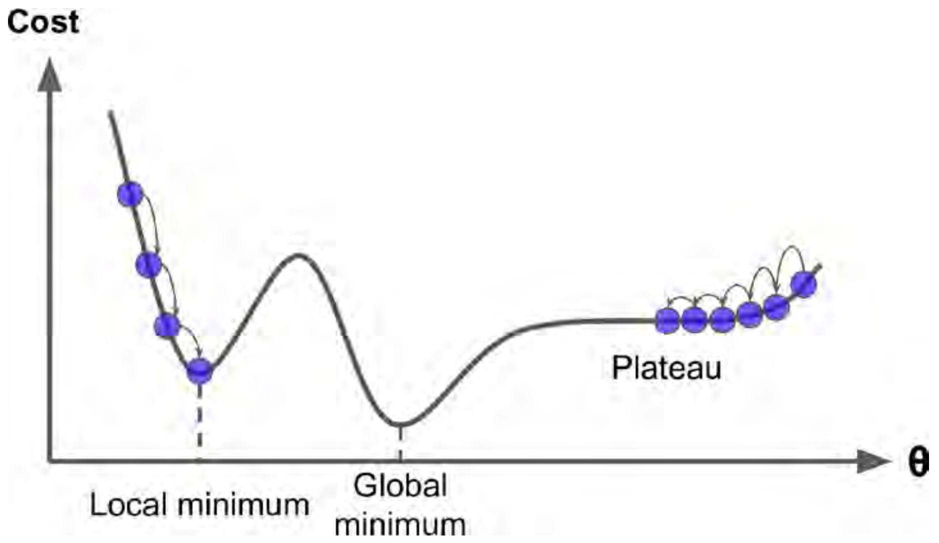


Figure 2.12: Convergence issues of the GD algorithm in a non convex function, GÉRON, 2019, chap. 4

There are some possible adaptions of the GD algorithm to overcome this convergence issues. Bhalley explains in his chapter about Neural Networks BHALLEY, 2021b, chap. 5 optimizations of GD in more detail like for example adding a momentum. Furthermore, Ruder also analyses different optimization algorithms including their challenges RUDER, 2016. BHALLEY, 2021b, chap. 5; GÉRON, 2019, chap. 4; RUDER, 2016

Variants of the Gradient Descent (GD) algorithm

There are different variants of the GD algorithm depending on the amount of training samples used to calculate the GD. When deciding which variant to choose, it is always a compromise between the accuracy of the parameter tuning and the computation effort. Figure 2.13 shows the different convergence paths of the three variants. RUDER, 2016

Stochastic Gradient Descent Stochastic GD calculates one update of the parameters based on one training sample z_i according to equation 2.6.

$$\widehat{\theta} = \theta - \eta \times \nabla_\theta \mathcal{J}(\theta, z_i) \tag{2.6}$$

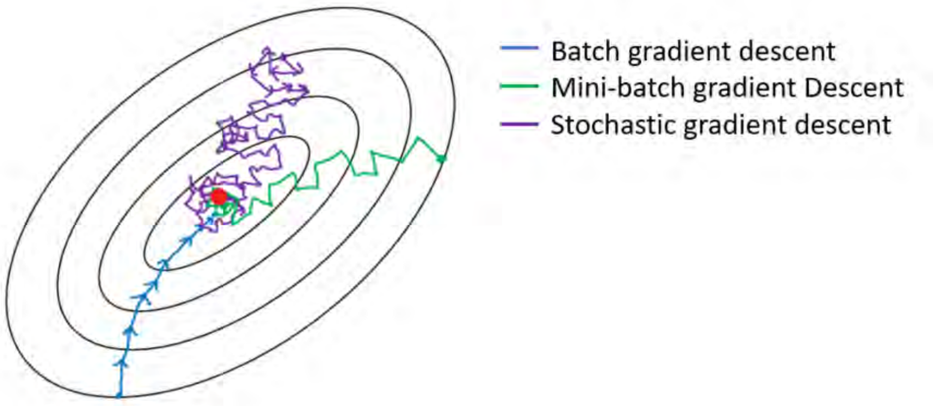


Figure 2.13: Convergence path within a convex function of the GD variants, DABBURA, 2017

The advantages are on the one hand the short computation effort, the little necessary memory space and the possibility of learning online which is explained in section 2.1.3. On the other hand the convergence to a local minimum is not guaranteed as it converges rather unpredictable according to figure 2.13. But due to the rather random iteration steps it is possible, that the Stochastic GD jumps out of a local minimum and converges to a global minimum. GÉRON, 2019, chap. 4; RUDER, 2016

Batch Gradient Descent This variant is also called Vanilla GD. It takes the whole training dataset z to calculate one iteration to the minimum of the objective function as described in equation 2.7.

$$\widehat{\theta} = \theta - \eta \times \bigtriangledown_{\theta}\mathcal{J}(\theta, z) \tag{2.7}$$

Taking for each iteration step the full training dataset makes the computation very slow and memory expensive. Further it is not possible to increase the dataset on the fly. But it converges regularly into the direction of the next local minimum. GÉRON, 2019, chap. 4; RUDER, 2016

Mini-batch Gradient Descent The Mini-batch GD is a mixture of both the Batch GD and the Stochastic GD. It takes only a small part containing n random samples of the training dataset for each iteration. It is assumed, that this gives an approximation of the whole dataset.

$$\widehat{\theta} = \theta - \eta \times \bigtriangledown_{\theta}\mathcal{J}(\theta, z_{i,\ldots,i+n}) \tag{2.8}$$

Accordingly, this increases the performance of the updates compared to Stochastic GD but does not need as much computation time and memory space as Batch GD. BHALLEY, 2021b, p. 221 ff; GÉRON, 2019, chap. 4; RUDER, 2016

Momentum

There exist different optimiser strategies for calculating the best gradient for updating the model parameter. If the objective function provides flat areas and also to avoid the oscillations of the gradient curve, adding an inertia to the gradient helps to overcome these areas and speeds up convergence. To accelerate the gradient, a fraction λ of the previous gradient update vector ν_{t-1} is added to the new gradient update vector ν_t. λ is normally a value between 0.0 and 1.0 and usually close to 1.0.

$$\nu_t = \lambda\nu_{t-1} + \eta\nabla_\theta\mathcal{J}(\theta) \tag{2.9}$$

The parameters θ are then updated by subtracting the update vector from the parameters; $\theta = \theta - \nu_t$. When the gradient is moving into similar directions the acceleration and therefore also the convergence speed increases further and further. As soon as the gradient moves in other directions, the speed decelerates and the update steps also become smaller. RUDER, 2016

Adam optimiser

The Adaptive Moment Estimation (Adam) optimiser computes individual learning rates for each parameter. These learning rates are adapted with each time step using exponentially decaying averages of the past squared gradients and also of the gradient itself. Two fraction parameters β_1 and β_2 are used for estimating the first momentum m_t being the mean of the gradient g_t and the second momentum v_t being the uncentered variance of the gradient g_t^2.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{2.10}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{2.11}$$

The two moments tend to be biased to zero as they are initialised with zero, especially, when the fraction parameters are close to one. They are corrected the following way:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.12}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.13}$$

Each parameter θ is updated using the individually adapted learning rate:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \tag{2.14}$$

For numerical purposes the bias value ϵ is added to the denominator. Normally it is a small bias in the range of 10^{-8}. RUDER, 2016

2.3 Convolutional Neural Networks

CNNs are a subtype of ANNs, that have a specific layer sequence for handling grid structures with local dependencies. They can basically be used for data of any spatial and/or temporal type. Thus it is perfect for dealing with image data, as objects within images are normally defined by a set of adjacent pixel and tend to be locally dependent. In this section the components and applications of CNNs are explained. AGGARWAL, 2018; BHALLEY, 2021a

2.3.1 Basic Structure

The layers of a CNN are connected similar to a feed forward network in an often sparse way. Thus, a feature from the previous layer is only linked to its spacial dependencies in the next layer in order to decrease the complexity by reducing the amount of parameters within the model. AGGARWAL, 2018

Each layer of a CNN consists of a tensor also called filter, containing parameters, that are trained in order to extract certain features. These Tensors are 3D and defined by their height, width and depth. In this case depth does not refer to the amount of layers in the network; instead it defines the amount of input channels of an image in the input layer or within the hidden layers the amount of filters used in each layer. AGGARWAL, 2018, BHALLEY, 2021a

In CNNs there are different ways of combining certain layer types. Figure 2.14 shows a common architecture for the task of classifying a handwritten digit. In the following these different layer types are described in more detail. AGGARWAL, 2018, BHALLEY, 2021a

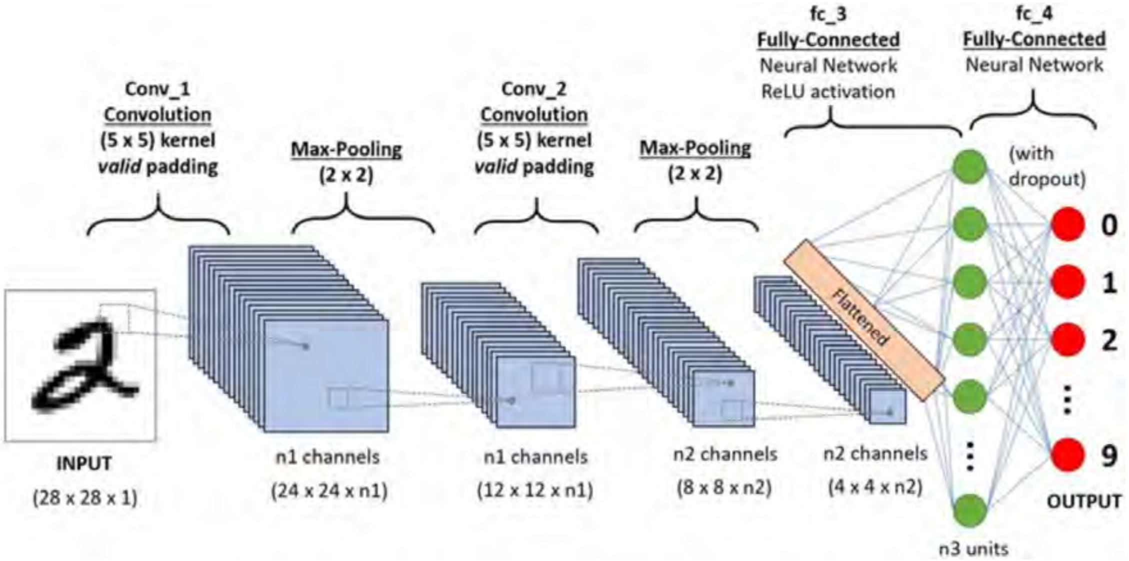


Figure 2.14: Architecture of a CNN for the task of classifying handwritten digits SWAMINATHAN, 2020

Input Layer

CNNs accept an image or feature vector as input. In the input layer one input node corresponds to one entry within the input map. These entries can either be one pixel in case of an image or one entry within the arbitrary feature vector. AGGARWAL, 2018

For each channel there exists one input map. This means, that for example for RGB images, there are 3 input channels with each channel defining the intensity of each base colour for each pixel. Greyscale images have only one channel and thus their depth is one. In the following layers the depth of the filters are the same as in the input data. This is visualised in figure 2.15. AGGARWAL, 2018; STEWART, 2019

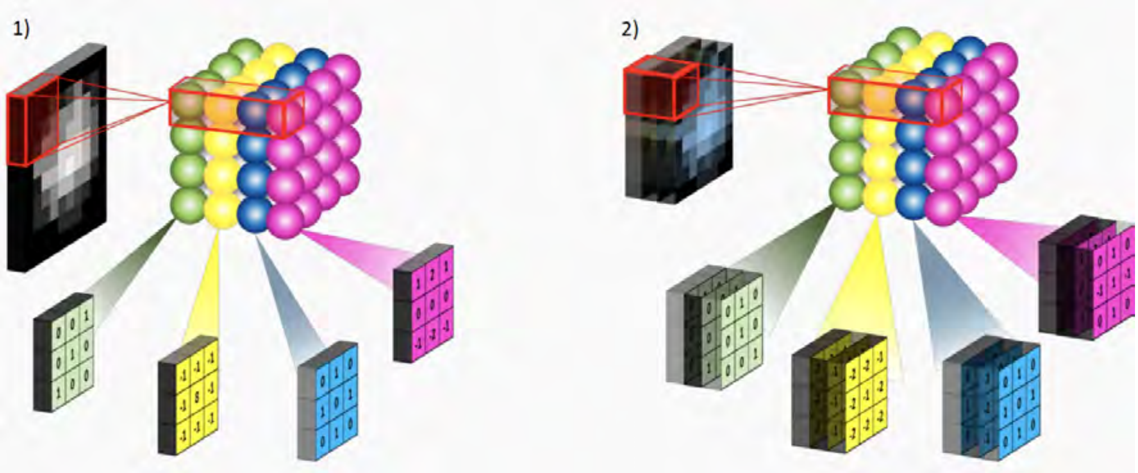


Figure 2.15: Differing depth of feature maps in Convolutional Layers depending on the amount of channels in the input image; 1) Greyscale and 2) RGB, STEWART, 2019

For CNNs the size of the input image should always stay the same, as the amount of weights in the fully connected layer depends strongly on the size of the data.

Convolutional Layer

Within a CNN there can be several convolutional layer containing different filters. These layers are responsible for the network being able to process the local correlation of pixel in the input feature map QIUHONG KE et al., 2018. With this knowledge the network is able to detect the presence of features within the input image and summarize these features into a feature map. AGGARWAL, 2018

Convolution is an operation between the feature map and a filter. Their depth must be the same but their size dimensions can differ. Mathematically convolution is the dot product between the full 3D filter and sections of the feature map. These sections are at defined spatial positions (i, j), so that the filter slides through every possible position in the feature map without overlapping the borders of the feature map. AGGARWAL, 2018; BHALLEY, 2021a; GÉRON, 2019

The parameters of the pth filter in the qth layer are described by the 3D tensor $W^{(p,q)} = [w_{ijk}^{(p,q)}]$, whereas i, j, k denote the position with regards to height, width and depth in the filter. The tensor has the size $FL_q \times FB_q \times d_q$. The 3D tensor $H^{(q)} = [h_{ijk}^{(q)}]$ stands for the feature map in the qth layer with the size $L_q \times B_q \times d_q$. The size of the next feature map is defined by the amount of executed dot products or in other words by the number of possible positions of the filter within the feature map in the previous layer. Thus, the height of $H^{(q+1)}$ is $L_{(q+1)} = (L_q - FL_q + 1)$ and its width being $B_{(q+1)} = (B_q - FB_q + 1)$. AGGARWAL, 2018; BHALLEY, 2021a; GÉRON, 2019

The bias term b_{pq} adds a value to the convolution operation for each filter p in order to for example adjust the brightness within an image. The convolution operation calculates each entry in the feature map of the next layer following equation 2.15. Convolution is visualised in figure 2.16 including the extra features padding and striding, which are explained in more detail in the following paragraphs. AGGARWAL, 2018; BHALLEY, 2021a; GÉRON, 2019

$$h_{ijp}^{(q+1)} = b_{pq} + \sum_{r=1}^{FL_q} \sum_{s=1}^{FB_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \begin{array}{l} i \in \{1, ..., L_q - FL_q + 1\} \\ j \in \{1, ..., B_q - FB_q + 1\} \\ p \in \{1, ..., d_{q+1}\} \end{array} \tag{2.15}$$

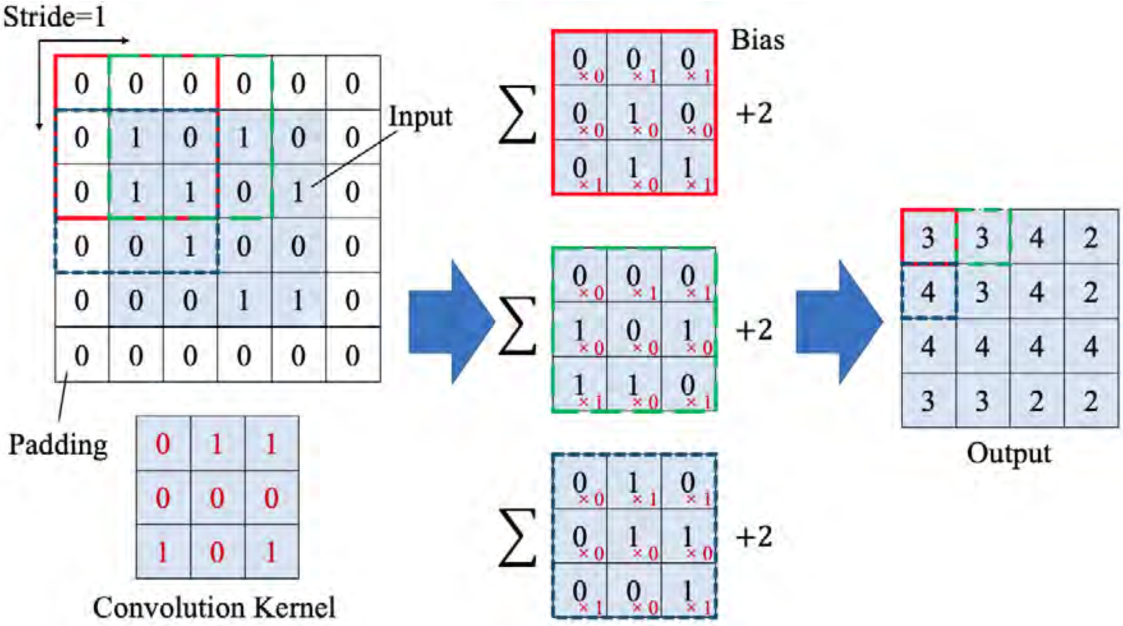


Figure 2.16: Convolution operation for one filter in different spatial positions with half-padding, SHI et al., 2021

Padding Convolution decreases the size of the feature map and thus, information from pixel close to the borders tend to get lost, as these pixel are under-represented within the following layer. Padding is a strategy to overcome this problem by adding several rows and columns at the borders with the pixel value zero. To have the same size of the feature map $H^{(q)}$ and $H^{(q+1)}$ in the following layer the height of $H^{(q)}$ has to increase depending on the filter size by $(FL_q - 1)$ and its width by $(FB_q - 1)$. This type of padding is called half-padding and is represented in figure 2.16. AGGARWAL, 2018; BHALLEY, 2021a

Strides Another way of decreasing the size of the feature map in the following layer is to use strides. A stride of $S_q = 1$ means, that the filter is applied to every possible spatial position within the feature map. If $S_q \neq 1$, the convolution is performed at the spatial locations $1, S_q + 1, 2S_q + 1, ...$ in both directions. Figure 2.16 uses a stride of 1. The higher the stride, the less memory is consumed but the less accurate is the representation and resolution. AGGARWAL, 2018; BHALLEY, 2021a

ReLU Activation function

ReLU stands for Rectified Linear Unit (ReLU) and is a piecewise linear function, that outputs all values smaller than zero as zero; $f(x < 0) = 0$ and returns the same input values, if they are greater than zero; $f(x >= 0) = x$. It's shape is plotted in figure 2.9e. After a Convolution layer the ReLU activation function is applied on each entry in the feature map similar to the classical ANNs. AGGARWAL, 2018

According to the paper written by Krizhevsky et al. CNNs using the ReLU activation function train much faster then when other activation functions are used KRIZHEVSKY et al., 2012. As ReLU proves to have lots of advantages with regards to speed and accuracy it is the most commonly used activation function within CNNs. AGGARWAL, 2018

The 'Leaky ReLU' activation function is a modification of the classical ReLU function. When dealing with values bigger than zero it still outputs the value; $f(x >= 0) = x$. To negative values a small slope coefficient s being determined before training is applied instead of outputting zero; $f(x < 0) = ax$. When dealing with poor gradients, for instance when training GANs, this activation function type can return better results. AGGARWAL, 2018

Pooling Layer

For decreasing the size of the feature maps after a convolution layer the pooling operation can be used. Pooling means, that squared grid regions of the feature map are summarized by applying a certain function. The grid region has a size of $P_q \times P_q$ and moves through the feature map with a stride size S_q, that is normally bigger than one. The resulting feature map $H^{(q+1)}$ has the same depth, a length of $L_{q+1} = (L_q - P_q)/S_q + 1$ and a width of $B_{q+1} = (B_q - P_q)/S_q + 1$. AGGARWAL, 2018; YINGGE et al., 2020

The most common pooling operation is called *max-pooling*. This method extracts only the maximum value within one grid region and copies it to the output feature map. Another more seldom used method is *average-pooling*, where the average value is calculated for each patch of the feature map. These pooling operations are illustrated in figure 2.17. Pooling Layers are useful for having an invariance to translation up to a certain degree, as the feature map stays the same for slight shifts within the input image. This helps to identify the same feature in different slightly changed images. AGGARWAL, 2018; YINGGE et al., 2020

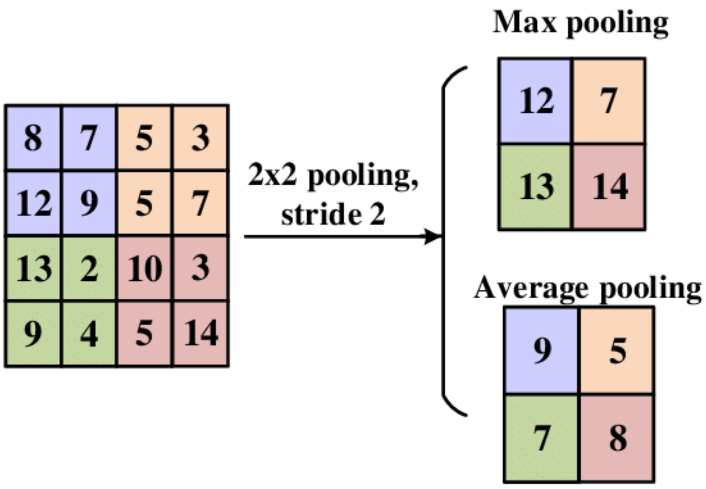


Figure 2.17: Output feature map after applying the Pooling Operation, YINGGE et al., 2020

Fully connected Layer

Fully connected Layers normally follow the convolution and pooling operations and behave like a regular feed forward network. The data arrives spatially clustered after undergoing the convolution process in form of feature maps. These feature maps for each image instance are converted into a vector in a flattening layer and are fed into the fully connected Layers. As these layers are so densely connected there are lots of parameters needed, that usually take the main processing time during training. AGGARWAL, 2018

The fully connected Layers are used for further processing tasks like classification. In figure 2.14 the fully connected layers classify the handwritten input digit and return the probability for this image being a certain digit. The output of the fully connected Layers represents the final output of the full network. To reduce the computation complexity as well as the amount of parameters of the network during training, the so called *Dropout* Training is used. Therefore a certain percentage of neurons are turned off randomly in each layer during training. In each Training iteration step other neurons are shut down. *Dropout* also helps to reduce the danger of overfitting. AGGARWAL, 2018; ALZUBAIDI et al., 2021; GARBIN et al., 2020; HAIBING WU and XIAODONG GU, 2015

Batch normalization

This is a regularization technique, that applies normalization on a mini-batch of data in each layer within the network. The input data batch is re-scaled and re-centered in each layer. On a mini-batch B of the size m taken from the training set, the mean μ_B and variance σ_B^2 of the batch are calculated as follows:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.16}$$

To normalize a d-dimensional vector with input values of $x = (x^{(1)}, ..., x^{(d)})$ each dimension gets its own mean $\mu_B^{(k)}$ and standard deviation $\sigma_B^{(k)}$ value. x is normalized to $\hat{x}$ with ϵ being a small constant to provide numerical stability.

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{(\sigma_B^{(k)})^2 + \epsilon}} \tag{2.17}$$

Batch normalization enables the use of higher learning rates, that decrease the amount of iterations needed for training and accordingly the computation time for training the network also decreases. Batch normalization adjusts the units for each batch and it chooses the mini batches randomly during training. This leads to an increase in noise during training. The noise in turn helps to regularize the network and reduces the problem of overfitting. ALZUBAIDI et al., 2021; GARBIN et al., 2020

Dropout

The dropout method was originally designed to avoid overfitting in feed forward neural networks. It is also a regularization method. The basic idea is to drop randomly (usually with a probability of 0.5) neurons in the layers during the training iterations. This means, that random input units are set to 0 with a defined frequency rate during each training step. For CNNs there exist more specific methods, where not only single pixel are dropped but whole regions, features or squares of pixel. Instead of the neurons, also connections can be dropped during training with a certain probability, which is called dropconnect. LABACH et al., 2019

Skip Connections

Skip connections provide an alternative path between the layers and accordingly for the gradient calculation during backpropagation. It means, that some layers are skipped within the network and the output is put directly into another layer as input. The gradient of early layers in a deep network can become very small, as due to the chain rule, the multiplication term for updating the parameters in early layers during backpropagation becomes large. This leads to small or even zero parameter updates in the early layers. ADALOGLOU, 2020

Information can be passed directly across the network between the input and output layer using skip connections. Basically, the skip connection concatenates two layer of the same dimensionality. In encoder-decoder structures like the UNet long skip connections are used for passing information in order to recover fine details in the prediction. Figure 2.18 shows the long skip connections between layers with the same dimensionality in the encoder and decoder part of a UNet and on the right the concatenated output of a layer and a skip connection. ADALOGLOU, 2020

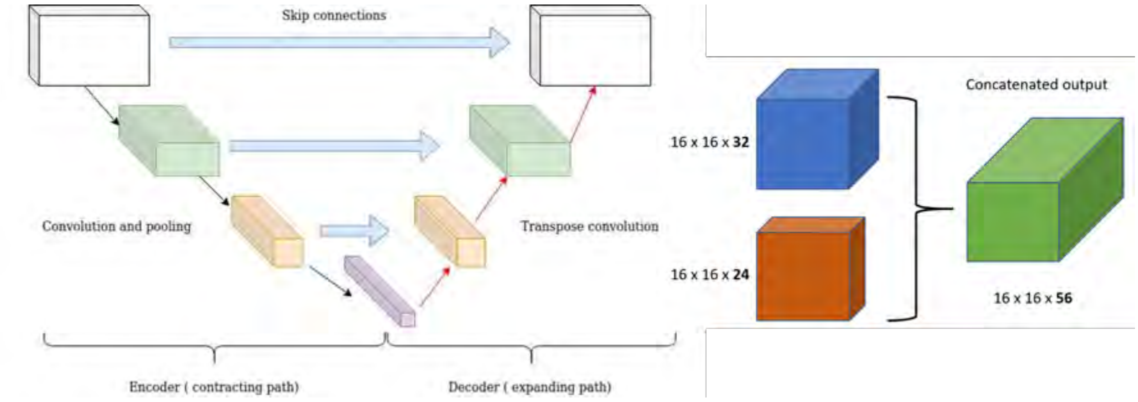


Figure 2.18: **Left:** Long Skip Connections between the encoder and decoder in a UNet structure; **Right:** concatenated output of a layer in the decoder with skip connections ADALOGLOU, 2020

2.3.2 Application Categories

The big advantage of a CNN is, that it can identify important objects by itself. This ability finds use in many different fields. CNNs are used for image or video recognition, speech processing and several computer vision tasks. In the following the five main application categories of CNNs are explained in more detail. These are also summarized in figure 2.19. ALZUBAIDI et al., 2021

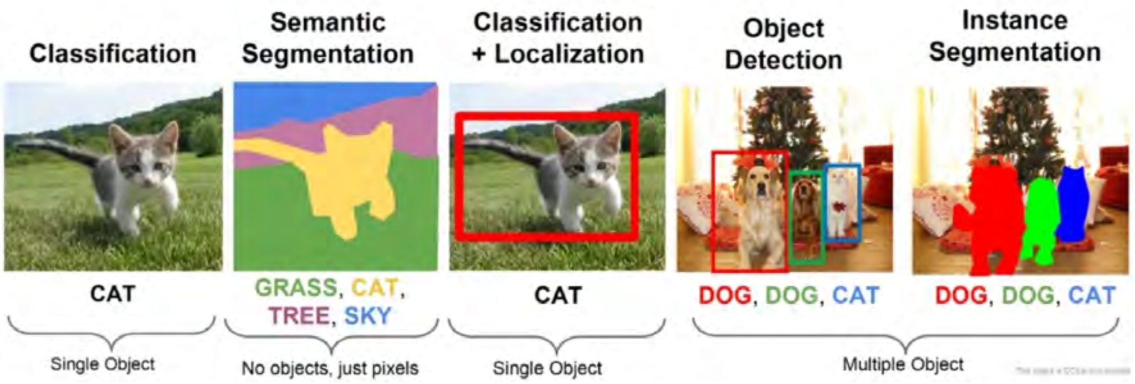


Figure 2.19: Common applications of CNN FRAJBERG, 10/30/2017

Classification Classification in context of ML and CNNs is a technique to retrieve the matching class of an object in an image according to its visual appearance. A neural network is trained to analyse for example the shapes, edges and colours within an image and determines the fitting class label out of a certain selection of classes the network was trained to recognise. Depending on the final activation function, the network either returns one class label or a probability distribution of all available classes within the network. Classification of videos is also possible, as videos are a sequence of images. In this case another dimension is included in the network filters besides the depth with regards to the amount of images in the video sequence. AGGARWAL, 2018; GÉRON, 2019

Localization Localizing an object within an image is done by finding a rectangular bounding box describing the scale and exact position of the object within the image. A common way of defining a bounding box is to have the 2D position of the top-left corner of the box and its width and height, leading to an unique identification of the box with four variables. Thus, Localization is a multivariate Regression task. Often, an object within an image is both classified and localized. AGGARWAL, 2018; GÉRON, 2019

Object Detection Object Detection is the localization and classification of a variable amount of objects within an image or video sequence. In order to reduce the computation time of the CNN, a region proposal method can be used beforehand. This method creates provisional bounding boxes within the image consisting of similar pixels. Then, the standard algorithm for classification and localization for one object is applied on each proposed region. AGGARWAL, 2018; GÉRON, 2019

Semantic Segmentation Semantic Segmentation is the labelling of each pixel within the full image. As illustrated in figure 2.19 the pixels are clustered and allocated into groups each belonging to a class without any information about their locations. The network does not distinguish between different instances of one class. FRAJBERG, 10/30/2017; JUNHO JO et al., 2019

Instance Segmentation In Instance Segmentation only the pixels of objects within an image belonging to certain classes are clustered and classified. Furthermore all instances in the image are labelled uniquely even though they might belong to the same class. Image segmentation methods are used for optical character recognition like for extracting and classifying handwritten text in printed documents. Other common applications are the modification of the background or of objects within images. FRAJBERG, 10/30/2017; JUNHO JO et al., 2019

2.3.3 Common Architectures

Depending on various applications, several CNN architectures have been evolved. In the following, two relevant architectures for this thesis are introduced.

U-Net The U-net architecture was originally created for biomedical image segmentation tasks, like for example finding tumors in lungs or in the brain. It is based on the structure of a fully convolutional network. The architecture is basically a combination of an encoding network followed by a decoding structure which is also called an autoencoder. RONNEBERGER et al., 2015

The encoder consists of convolution blocks followed by maxpooling downsampling layers, that are used for extracting features from the input image at different levels of detail. Afterwards the decoder projects the extracted features from the encoder onto the pixel space to get a dense pixel classification in a high resolution. Upsampling, concatenation and convolution operations are used during decoding to achieve the final resolution and classification of each pixel. The architecture is visualised in figure fig. 2.21. It is a useful network for semantic segmentation tasks and also copes well with a rather small amount of training data, when several data augmentation steps are applied to the available data. RONNEBERGER et al., 2015

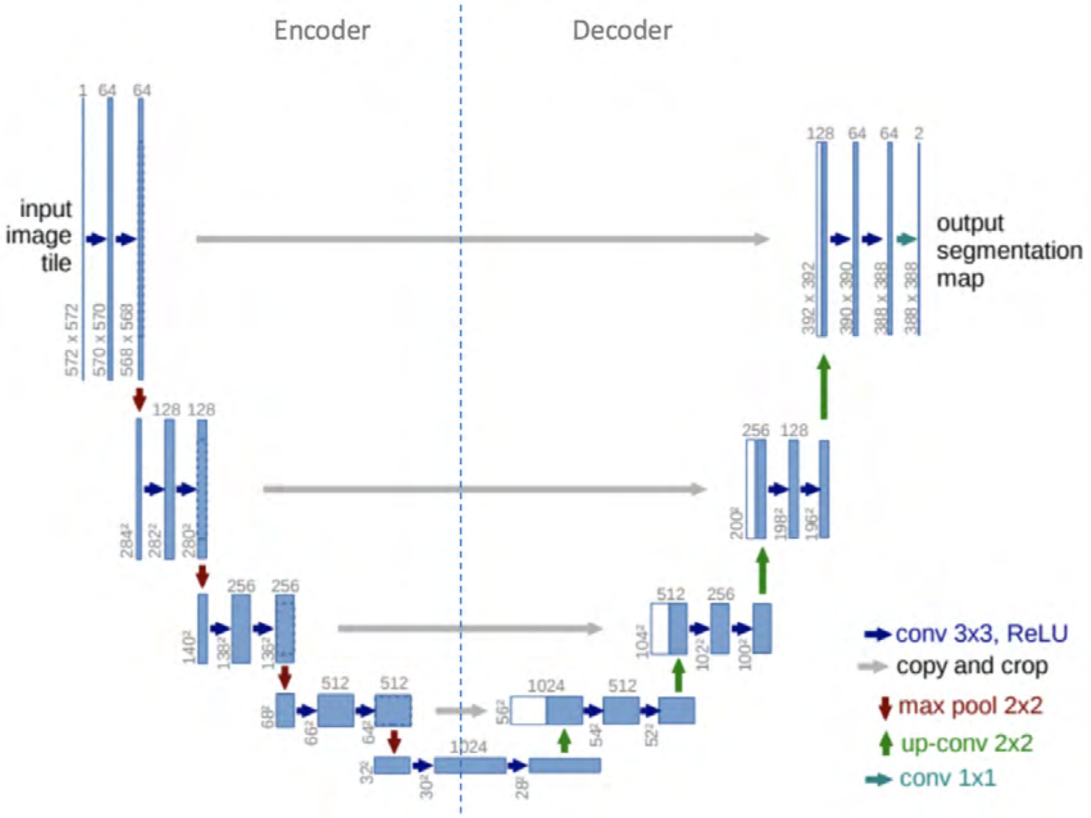


Figure 2.20: Architecture of a U-Net used for semantic segmentation tasks. RONNEBERGER et al., 2015

Convolutional PatchGAN The PatchGAN cuts down the image into smaller sections or patches in several pooling layers. Convolution and batch normalization layers retrieve information from these down-sampled patches and combine the information into one final output using convolution. Thus, it is a useful network for retrieving local information on a certain level of detail and finally combine it into one final output. ISOLA et al., 2016

The model assumes independence between pixels, that are separated by more than the patch diameter. Accordingly, the size N of the patches has a great influence on the detection of features within an image. This network structure can be used as Discriminator within GANs for detecting, if an image is real or fake. ISOLA et al., 2016

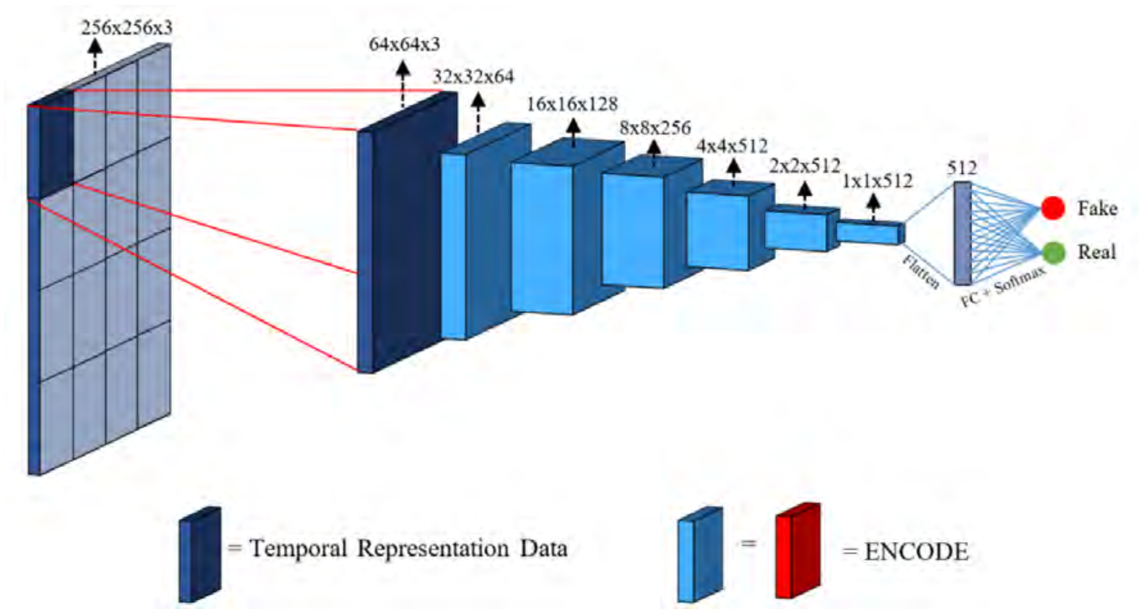


Figure 2.21: Architecture of a convolutional PatchGAN network used for the classification of images being real or fake. GANOKRATANAA et al., 2020

2.4 Generative Adversarial Networks (GANs)

Since their publication in 2014 Generative Adversarial Network (GAN) architectures are another recently booming type of ANNs commonly used for semi-supervised and supervised learning tasks CRESWELL et al., 2018; LI et al., 2021. A GAN consists of two separate networks, the Generator and the Discriminator. These networks work together in order to create new data samples, that are as close to the reality as possible. Because of learning indirectly from data, GANs provide completely new ways of predicting data with an improved ability to generalise in comparison to CNNs. GANs are applied in various image and vision computing tasks and are also useful in speech and language processing. The following section describes the basic architecture and provides a closer look into certain applications. WANG et al., 2017

2.4.1 Basic Architecture

Within a GAN two separate deep neural networks compete against each other. This idea is inspired by a two player zero-sum game, where one player wins only the equal amount of the other players losses. In case of a GAN one player denoted as Generator $\mathcal{G}$ creates new samples out of an input noise as close to the realistic data distribution as possible. The Discriminator $\mathcal{D}$ being the other player tries to identify, if a sample is real or fake. $\mathcal{D}$ gets both real samples from the training data as well as the synthetically produced samples from the Generator. Its goal is to learn to classify both samples as either real or fake as accurately as possible. CRESWELL et al., 2018; LI et al., 2021; WANG et al., 2017

Both networks are trained simultaneously in an alternating manner during training. When $\mathcal{G}$ is trained and updated, $\mathcal{D}$ is fixed and the other way round. The Generator produces a sample, which is then fed into the Discriminator with the label 'fake'. According to the output of $\mathcal{D}$ the parameters within the Generator network are updated. When the predictions of the Discriminator approach to 0.5 the optimal model for generating realistic data is produced, as it is not possible anymore for the Discriminator to differ between fake and real data. CRESWELL et al., 2018; LI et al., 2021; WANG et al., 2017

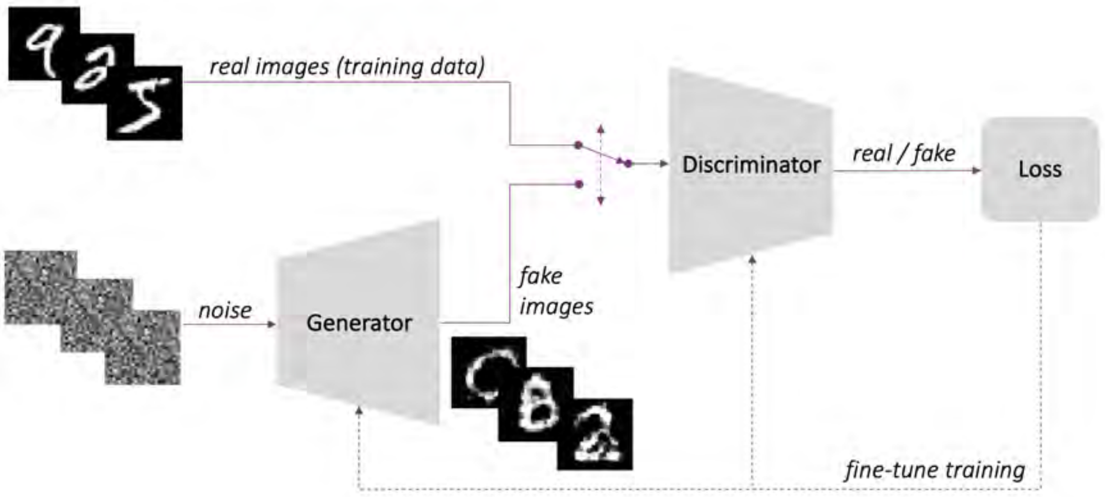


Figure 2.22: Basic architecture of a GAN using the example of creating handwritten digit images CHRISTINA KOURIDI, 2019

Objective Function The Generator $\mathcal{G}$ creates a mapping function to translate the prior data distribution vector z to an output y; $\mathcal{G} : z \rightarrow y$. Whereas the Discriminator $\mathcal{D}$ outputs a scalar value estimating the probability, that a sample comes either from the training dataset x being real or is a fake output created by the Generator y. The parameters of $\mathcal{G}$ are twitched in order to minimize $log(1 - \mathcal{D}(\mathcal{G}(z)))$ and for $\mathcal{D}$ to keep $log\mathcal{D}(x)$ as small as possible. As $\mathcal{G}$ and $\mathcal{D}$ are trained simultaneously, the objective function for the full network results in the following equation: ISOLA et al., 2016; MIRZA and OSINDERO, 2014

$$\min_{\mathcal{G}} \max_{\mathcal{D}} \mathcal{L}_{GAN}(\mathcal{G}, \mathcal{D}) = \mathcal{L}_x[log\mathcal{D}(x)] + \mathcal{L}_z[log(1 - \mathcal{D}(\mathcal{G}(z)))] \tag{2.18}$$

It acts like a 2-player min max game, where $\mathcal{G}$ tries to minimize the objective function and thereby competing against $\mathcal{D}$, that tries to maximize the objective function.

2.4.2 Image to image translation

The application of translating an image into another image is an important task in many problems belonging to the fields of graphics, vision and image processing. The basic task is to map the pixel of an input image to pixel of an output image and thereby fulfilling various translation tasks like for example showing a black-white image in colours or converting a scene, shot by daylight, at nighttime. Figure 2.23 shows some application examples for image to image translation. ISOLA et al., 2016

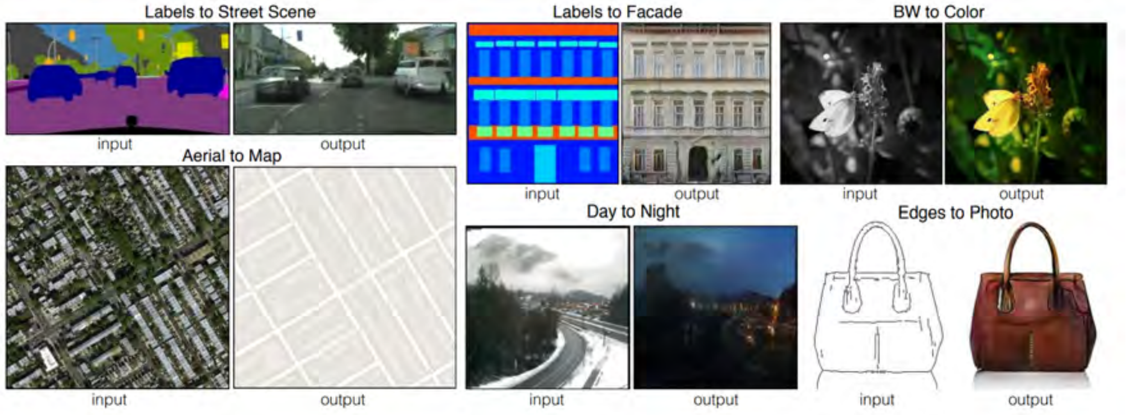


Figure 2.23: Application examples for image to image translations ISOLA et al., 2016

Pixel Classification vs Pixel Regression When images are mapped into another image each pixel is translated by receiving a certain prediction value. This pixel-value can either be obtained by classification, which means, that it gets a discrete class label, that is predicted with a potentially continuous probability for a certain class. Or, the value can be predicted using regression out of a continuous class space. The result is a discrete value, that can also be in the form of a class label. ISOLA et al., 2016

Unstructured vs Structured image An unstructured image is considered as completely pixel independent. When translating the image, the prediction takes place pixel wise without considering the nearby pixel. The opposite is a structured image. There, a bunch of pixel is combined and dependent from each other. Structured losses can be calculated and penalized leading to a network, being able to understand the context of pixel within an image better. A conditional Generative Adversarial Network (cGAN) architecture can be used for structured images. ISOLA et al., 2016

2.4.3 Conditional Generative Adversarial Networks (cGANs)

In a conditional Generative Adversarial Network (cGAN) additional information v is both available for the $\mathcal{G}$ and the $\mathcal{D}$ models. The information v can be class labels or other data. It is given to both the $\mathcal{G}$ and the $\mathcal{D}$ as additional input layer. The objective function changes to the following equation: ISOLA et al., 2016; MIRZA and OSINDERO, 2014

$$\min_{\mathcal{G}} \max_{\mathcal{D}} \mathcal{L}_{cGAN}(\mathcal{G}, \mathcal{D}) = \mathcal{L}_x[log\mathcal{D}(x|v)] + \mathcal{L}_z[log(1 - \mathcal{D}(\mathcal{G}(z|v)))] \tag{2.19}$$

A cGAN learns a structured loss. This loss punishes full structured parts within the generated image, that deviate from the corresponding part in the target image. MIRZA and OSINDERO, 2014

pix2pix Isola et al designed a cGAN architecture called 'pix2pix', applicable for pixel wise image to image translation tasks. They have input images x as well as the random noise vector z as additional information. The Generator should be trained to map the input image to an output image y; $x \to y$. The noise z is not provided as an initial vector to the Generator but is instead replaced by dropout on several layers during training and testing. ISOLA et al., 2016

To encourage the Generator to - besides fooling the Discriminator - create images being also close to the target image, the L_1 distance is used as well for updating the parameters of $\mathcal{G}$. The L_1 Loss function measures the absolute distance between the generated image and the target image and is calculated according to eq. (2.20). The L_2 distance would create more blurring; it is accordingly not applied in this case. ISOLA et al., 2016

$$\mathcal{L}_{L1}(\mathcal{G}) = \mathcal{L}_{x,y,z}[||y - \mathcal{G}(x,z)||_1] \tag{2.20}$$

The objective function of $\mathcal{D}$ stays the same as in other cGANs. The objective function for updating the parameters within the Generator including the L_1-Loss depending on the weight parameter λ is calculated as follows: ISOLA et al., 2016

$$\mathcal{G}^* = \arg\min_{\mathcal{G}} \max_{\mathcal{D}} \mathcal{L}_{cGAN}(\mathcal{G}, \mathcal{D}) + \lambda \mathcal{L}_{L1}(\mathcal{G}) \tag{2.21}$$

Sigmoid Cross Entropy Loss The generator loss is calculated using the sigmoid cross entropy loss of the generated image, that has been classified by the discriminator and an array of ones, which is equivalent to the discriminator classifying all pixels as real. It calculates the loss independent of the other vectors or pixels in the image. The sigmoid cross entropy loss, also called binary cross entropy loss, deals with binary classification problems. It differs between real ($= 1$) and fake ($= 0$) pixel patches and deals thus with a binary classification problem of two classes. The sigmoid activation function $f(s_i)$ is combined with the cross entropy loss L_{CE} and is calculated as follows: GÓMEZ, n.d.

$$f(s_i) = \frac{1}{1 + e^{-s_i}} \tag{2.22}$$

Using the activation function, the binary cross entropy loss is derived:

$$L_{CE} = -\sum_{i=1}^{\#C=2} t_i \log(f(s_i)) = -t_1 \log(f(s_1)) - (1 - t_1) \log(1 - f(s_1)) \tag{2.23}$$

s_1 is the score and t_1 the label of the groundtruth for the first class. The score of the other class is $s_2 = 1 - s_1$ and for $t_2 = 1 - t_1$ as there are only class values of one or zero in the binary case. GÓMEZ, n.d.

2.4.4 Training a GAN

As the architecture of GANs is a very recent field of research, there is not yet a common recipe for tuning hyperparameters in a way, that always works. It is useful to play around with different batch sizes and learning rates. During training, the network's learning can be divided into two periods. In the first period, the GAN model learns very fast the most common characteristics and features within the image. Thereby having a high learning rate. In the second state the learning is rather slow and also the loss curve seems rather steady. In this state, the network learns the small and deeper details of the image translation. The parameter update in the model is really fine in order to not destroy the already learned features from the first period. GAN models do usually not converge. When training is successful, an equilibrium between the discriminator and the generator is found. Therefore it is hard to predict, how many epochs might make sense. Da Silva, n.d.

Loss Curve The Loss Curve of a GAN measures the training progress of the Discriminator and the Generator. The curve is created by calculating the losses during training and plotting it against the training step. A typical problem of training GANs is, that initially the generator is not able to generate images, that are predicted as true by the discriminator. A reason can be a bad dataset base. Saxena and Cao, 2020

Also, there can be strong oscillations in both loss curves. If either the discriminator or the generator loss gets very low, it outperforms the other part. When the discriminator always classifies the generated image as fake, there are no updates in the loss function for the generator and it stops training. This phenomena is called vanishing gradient. Saxena and Cao, 2020

2.4.5 Evaluation Metrics

Evaluation metrics are used after training a model in order to measure the performance of a ML model. There are quantitative measurements and also qualitative measurements. Similar to the Loss function during training, that measures the training progress, quantitative evaluation metrics are functions, that calculate a certain score based on certain distributional characteristics. Saxena and Cao, 2020

Common scores for rating GANs are for example the Fréchet Inception Distance (FID) or the Inception Score (IS). Even though there already exist some quantitative metrics for evaluating GANs, it is still a complex task, as GAN applications vary a lot. Visual evaluation is accordingly still the most common way of analysing the performance of a GAN, even though it is subjective, needs lots of time and captures no distributional characteristics. Saxena and Cao, 2020

2.5 Challenges of ANNs

When creating, training and applying ANN models, there are certain aspects, that need to be taken into account during every model creation. Important challenges are; a mindful selection and preparation of the dataset, optimising hyperparameters in an efficient manner and adapting the architecture of the network in order to avoid overfitting and underfitting.

2.5.1 Feature Preprocessing

The performance of ML models depends strongly on the quality and quantity of the data. Thus, a careful selection, preparation and processing of the dataset is essential. The following steps need to be taken into consideration when preparing the data.

Scaling the data In an ANN it is useful, that all features and values of a dataset have a similar scale. Otherwise converging would take much longer and the learning function would be more sensitive to some features than to others. The higher the value of a feature in comparison to other features, the more influence would it have on the error function. A similar scale of the features could for example be achieved using feature normalization. Also, every input sample must have the same size, as the network is usually adapted to a certain amount of input features. AGGARWAL, 2018

Deletion of outliers in the data A dataset must include consistent samples using always the same class labels/symbols/names. Otherwise, the network gets confused and is not able to learn. Thus, outliers must be deleted in advance. Also, samples or features not containing important information for the network are unnecessary as they increase computation time during training, use more memory space and confuse the model by distracting from actually important relationships. AGGARWAL, 2018

Data augmentation When working with data in image format like for instance mapping, segmentation or classification tasks, there are several options for augmenting the images. Depending on the data type, format and the problem normally only some augmentation steps make sense. Data augmentation can help increasing the dataset, when the amount of data is not sufficient for training a model. This reduces costs for creating and labelling data. GÉRON, 2019

Also, it makes data more robust against translation, scaling or rotation invariances and thus increases the generalization ability of the model. Common augmentation steps are colour modifications, for example darkening, brightening, greyscaling, changing the contrast; also rotation, padding, rescaling, flipping, translation, cropping, noise adding, random erasing and zooming are augmentation steps. GÉRON, 2019

2.5.2 Overfitting and Underfitting

Overfitting and underfitting are two common problems in context of neural networks. These are strongly dependent on the quality and amount of training data. Overfittig means, that a network model can perform well in predicting the training data, but if it tries to predict unseen test data, the performance can be bad. Thus, the model is not able to generalize well to new data, as random pattern in the data were also encoded within the model parameters. Increasing the amount of training data can help against overfitting and improve the generalization power. Géron, 2019

But, when using a simple model structure and a lot of data, the model might not capture complex relations, which is called underfitting. Increasing the model complexity leads to a capturing of more complex and detailed correlations between the input and target features of the model and can also facilitate overfitting, with too less data. Figure 2.24 visualises the problem of over- and underfitting. Alzubaidi et al., 2021

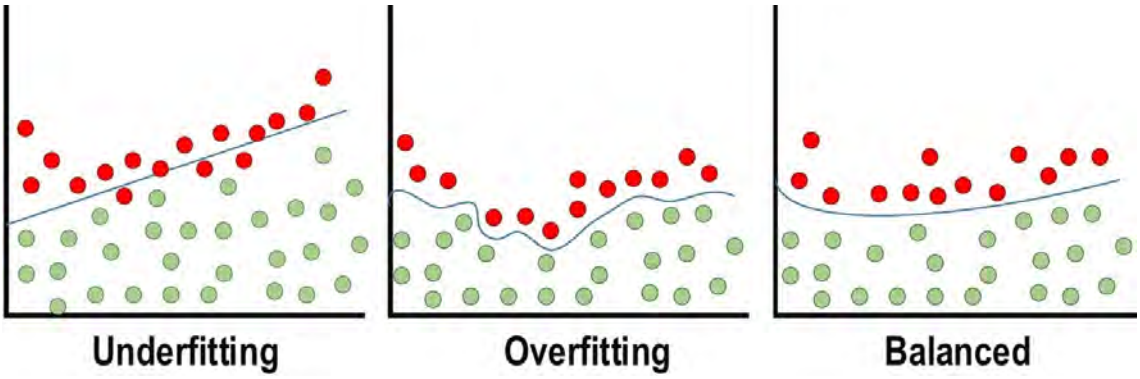


Figure 2.24: Visualization of the over- and underfitting problem Alzubaidi et al., 2021

2.5.3 Hyperparameter tuning

Hyperparameters are used in ML to control the learning process. Thus, they need to be selected carefully depending on the problem, the model type and the dataset.

Batch size The batch size is the amount of data samples, that is propagated through the network before a single forward and backward pass is done, in other words before the model parameters are updated. It makes sense to always choose a batch size to the power of two in order to use the Graphics Processing Unit (GPU) best.

Also, the system hardware might limit the batch size to a rather small number, if data samples are big. The batch size has an influence on the accuracy of the network and the time until convergence. A smaller batch size needs longer for training, as each step in GD might be less accurate and decelerates the convergence, but can still converge faster overall than a large batch. Also, a rather small learning rate is useful in combination with a small batch size, so that it is not overshooting the minima due to the variances. Larger batches might perform better in reaching the optimal minima and help the model to generalize better, but can also loose important relations between the input and target. KANDEL and CASTELLI, 2020

Dataset shuffling and Buffer size To feed the samples from the dataset in different orders into the network after each epoch, the dataset can be shuffled. Using a buffer size as big as the amount of datasamples or larger, the data samples get shuffled uniformly in a completely random way. A buffer size of one means no shuffling at all. The choice of the buffer size is depending on the dataset. MARTIN ABADI et al., 2015

Epochs An epoch is finished, when all training data has been shown to the network once and the parameters of the model were fitted. If there are too less epochs, the model is not trained enough on the dataset and is thus not able to generalize well and translate important details. When there are too many epochs, the model is overfitting and also not able to generalize well on unseen data. Thus, the number of epochs must be selected carefully. The optimal amount of training epochs is achieved, when especially the loss curve of the validation data has achieved minimal values.

Learning Rate The learning rate is the step size, the parameters are adapted in the direction of the GD. The smaller the learning rate, the more reliable is the training process, but it might take many iteration steps until convergence and thus a long iteration time. A high learning rate can lead to overshooting and no convergence at all. The optimal learning rate decreases with every learning step, when approaching the minima. With increasing batch size, also the initial learning rate should be higher. AGGARWAL, 2018

2.5.4 Network architecture adaption

To get the best fitting network architecture it is best to first find an already existing network from a similar problem and reuse and fit it to the new problem.

Amount of Nodes per Layer The input and output layers must have the same amount of nodes as there exist input and output features. A good selection of the number of neurons in the hidden layers is very important. On the one hand, too less neurons result in underfitting, so that the network is not able to detect more complex relationships. On the other hand, too many neurons lead to overfitting, as there are too many nodes for processing the information from the data. The model will perform perfectly well on training data, but is not able to generalize on unseen data. Also, the training time increases when more neurons need to be adjusted within every training step.

Amount of Hidden Layers For simple tasks, a rather shallow network with sometimes just two layers can be enough. Also, when the dataset is rather small, a shallower network might perform better. Overfitting is avoided and the computation time for training is faster.

For more intricate tasks like object or handwritten character recognition, a deeper network with more layers helps the model to detect more complex features and relations.

Type of layer When being no expert in creating neural networks, it is best to take the network architecture from a similar problem as basis and experiment by changing only small parts bit by bit. Still, some layer types usually come together with other layer types. Also, the selection of the types depends strongly on the data input and output format and the problem task. Educated guessing is essential for modifying the model in a helpful way.

Chapter 3

Related work - AI in the construction Industry

With the uprising of BIM and the accompanying digitization of several steps of a buildings life-cycle summarized in one model, a translation of the reality into a computational model becomes increasingly important and a linkage between both. Thus, the today's construction industry starts to rely more and more on the help of AI in various sectors. The objective is to let machines perform time consuming translation and calculation tasks to support the construction process and avoid or even correct human made errors.

Also, AI enables an easier passing of information between the planning office, the construction site and the digital model, as drones for instance can send regular status updates from on-site. In the office computers can restructure automatically the construction process on demand depending on the evaluation of the images provided by the drones.

This chapter introduces several projects related to the task of using DNNs to simplify analytical problems and the communication between construction workers on-site, civil engineers in an office and the digital model of a building. It also dives into the problem of designing a possible processing pipeline for automatically translating handwritten changes on drawings into the according BIM model by introducing paper, that provide solutions for certain processing steps.

3.1 Transfer of modifications on drawings into the BIM model

Even though the construction industry makes more and more use of BIM technologies, drawings will still be necessary in many designing steps in the coming years. Thus, an important field of recent research is an effective matching of geometric elements of a drawing to its digital twin in the 3D BIM Model at different Level Of Detail (LOD). And going one step further to transfer automatically and concurrently changes in the model to digital drawings and vice versa and accordingly keeping all excerpts in the same version. In this section symbol spotting in floor plans as well as two methods for linking a drawing to its digital twin are explained. Further a paper for the automatic translation of pixel based floor plans into vector based descriptors is presented.

3.1.1 Symbol Spotting in Architectural Floor Plans

Architectural floor plans provide a 2D representation of a horizontal cut through a building. The plans capture the arrangement of the basic architecture of one level of the building such as walls, openings and room constellations. Further, additional feature elements like windows, stairs, doors and sometimes furniture are illustrated. A *symbol* is a specific mark, shape or character, that is used to represent uniquely and in a simple way a certain object, action or idea.

Symbol Spotting is a way to both localize and classify a symbol within an image using a DNN and is the successor of symbol recognition. The computational complexity is decreased as the previously used full recognition methods are not necessary any more to that extent.

Rezvanifar summarizes in his paper the state of the art of Symbol Spotting in context of Architectural floor plans REZVANIFAR et al., 2019. It is an essential preliminary task in converting floor plans automatically into other representations and an important step for the digital interpretation of floor plans ZIRAN and MARINAI, 2018.

The challenge of symbol spotting in floor plans is according to Rezvanifar, that floor plans usually are large images with many symbols and further contain lots of clutter like an overlapping of symbols or text. Thus it is not easy for the model to spot symbols within the context of a floor plan. REZVANIFAR et al., 2019

Rezvanifar considers a symbol spotting method to consist of three main levels. The first level is called descriptor and defines the way the query symbols and the input floor plan documents are described. Geometric features in the documents can be defined pixel-based or vector-based, which means, that a symbol is defined according to its combination of lines, arcs or certain segments. The second level comprises various methods for matching and locating symbols in the plan as fast as possible with respect to certain challenges. And the final level deals with the performance evaluation and the further processing of the output. Figure 3.1 summarizes the three processing phases of a symbol spotting framework and points out methods for each level. REZVANIFAR et al., 2019

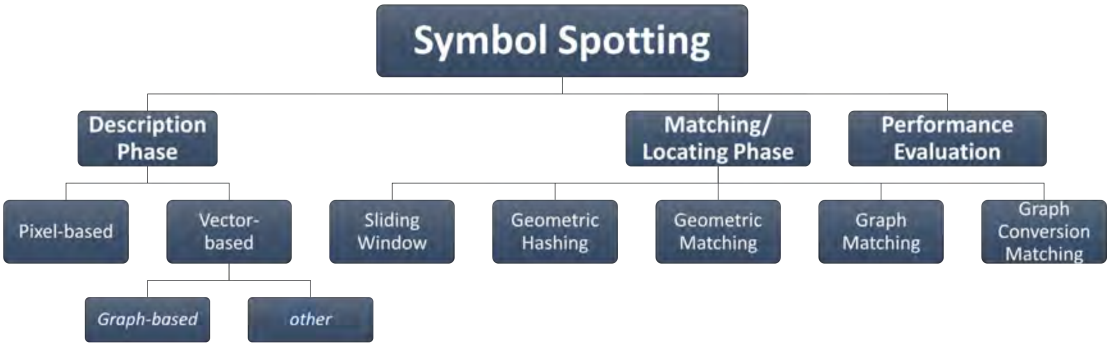


Figure 3.1: Overview of the three phases of a symbol spotting framework and useful methods in context of symbol spotting in floor plans REZVANIFAR et al., 2019

In the following, some applications of Symbol Spotting tasks are presented:

Segmentation Zeng et al. present a multi-task network for detecting pixel wise walls, windows and doors within floor plan images. This deep learning model is also able to classify, if pixels in floor plans are located inside or outside a house and further to classify the room type, each pixel belongs to. Accordingly this model solves pixel wise image segmentation tasks and belongs to the field of symbol spotting. The hierarchical structure of the full multi task floor plan segmentation model is illustrated in figure 3.2. ZENG et al., 2019

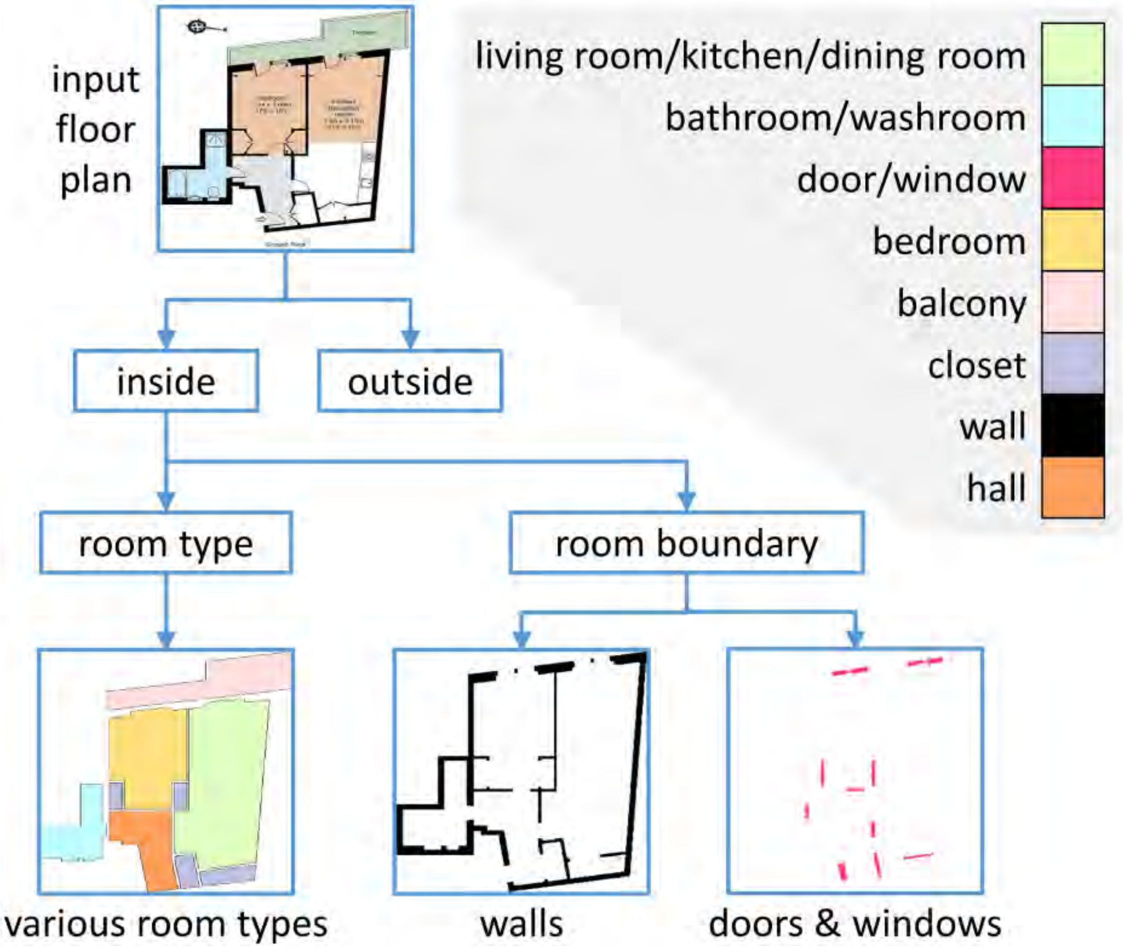


Figure 3.2: Hierarchical structure of a deep learning based multi task model for segmentation tasks within floor plans. ZENG et al., 2019

Object detection Figure 3.3 shows examples of two different floor plans including furniture symbols, where each symbol has been classified and localized with a bounding box using a Symbol Spotting approach based on object detection. According to Rezvanifar, the method of Symbol Spotting comprises a collection of query symbols. In case of this paper, these symbols are used as comparison elements to locate and classify similar symbols correctly within an input floor plan. REZVANIFAR et al., 2019

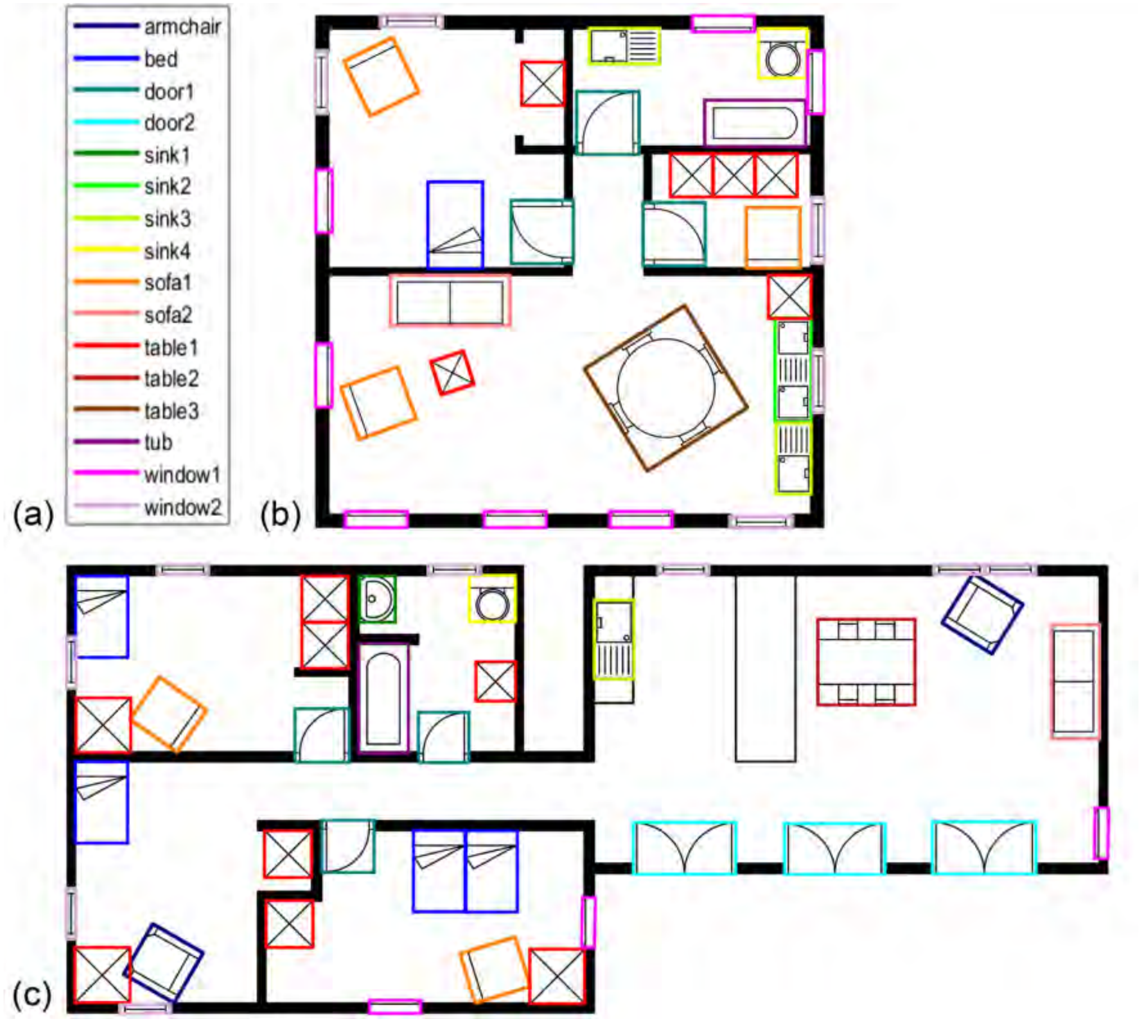


Figure 3.3: Example of a Symbol Spotting approach on architectural floor plans using different colours to classify a symbol and bounding boxes for localization Rezvanifar et al., 2019

Image mapping The concept designed within the scope of this thesis introduces a method for translating handwritten symbols from pixel-based input floor plan documents into its digital representation in pixel-based output floor plan documents. An artificial model is proposed, that is able to automatically spot handwritten symbols, internally classify them and translate the corresponding handwritten pixel into the matching digital symbol representation, predicting an output floor plan with the adopted changes. Accordingly, this thesis provides an extension and specific application of the methods for symbol spotting in architectural floor plans proposed in Rezvanifar et al., 2019.

3.1.2 Automated registration of drawings to BIM models

A solution for concurrent automated updating and consistency checking between construction drawings and the corresponding BIM model is necessary to obtain the error-free coexistence of both. In the following, two methods for the registration of drawings to its according BIM model are introduced. These can be used to check the similarities and disparities between both. TRZECIAK and BORRMANN, 2018

Also, this registration methods can provide an important step for enabling the transfer of modifications in drawings into the BIM model as they link features in the drawing to matching geometric features in the model. Section 4.3 explains, how the registration of drawings can be embedded into a full pipeline for translating handwritten symbols on drawings into the BIM model.

Extended Geometric Hashing

Trzeciak proposes a "drawing-to-model registration system", that "focuses on feature-based shape matching and uses knowledge-based extended geometric hashing as the method for measuring similarity and correspondence establishment between a query drawing and a set of cross-sections [...] derived from a 3D BIM model" TRZECIAK and BORRMANN, 2018. The processing pipeline is visualised in figure 3.4.

As a first step 2D cross-sections are extracted from the 3D BIM model by cutting horizontally through each level at a height of 1.20 m above the base of each level to reduce the problem to a 2D - 2D comparison and a manageable amount of cross-sections to compare the query drawing with. Secondly both in the drawing as well as in all cross-sections the geometric features are extracted using a vector based representation and stored in a hash table. As a third step based on a similarity measurement the hash tables are matched and the best voted cross-section is used for further aligning the drawing to the model. In the final alignment phase, the scale, the rotation and the translation vector is detected based on the difference between the drawing and the best fitting cross section. TRZECIAK and BORRMANN, 2018

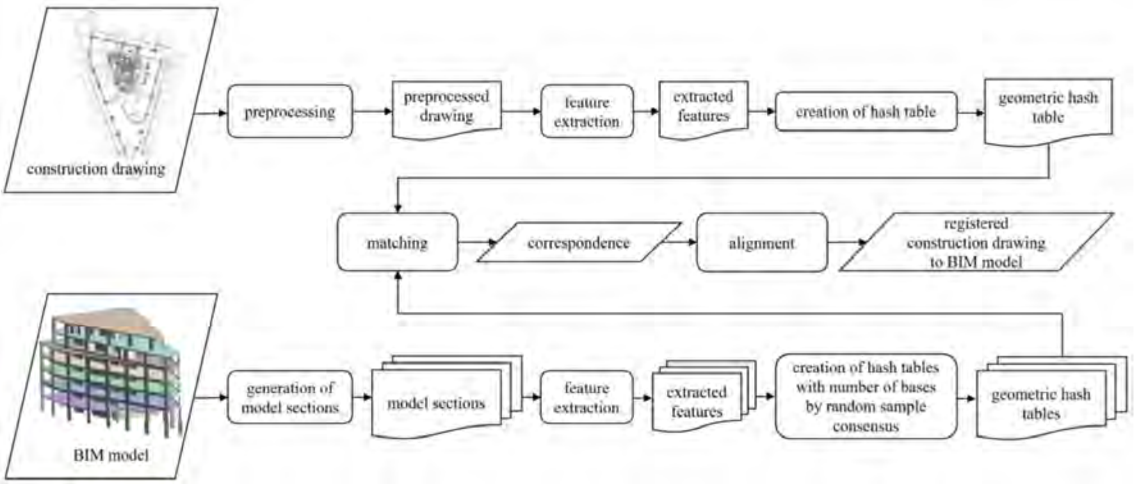


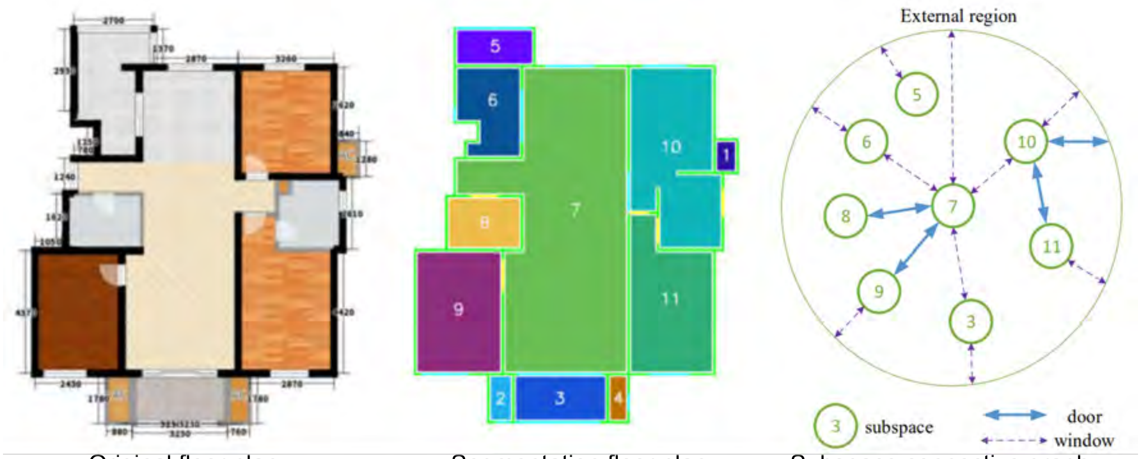
Figure 3.4: Pipeline for the drawing to BIM model registration system using geometric hashing TRZECIAK and BORRMANN, 2018

Registration Container

Another opportunity for linking drawing elements to the elements in the digital model are registration containers. These containers are intended for a bidirectional navigation across the drawing and model elements and provide an access to non geometric information from the drawing. Borrmann et al. propose four different linking mechanisms to connect drawings in svg format with the according Industry Foundation Classes (IFC) instance in the model using the IFC GUIDs. Borrmann et al., 2021

3.1.3 Vectorization of Floor Plans Based on EdgeGAN

Dong et al. have designed an EdgeGAN network, that is able to extract the vectorised representation of 2D floor plans. Structural, functional and furniture elements are detected, classified and localised. These feature element information is stored tree wise in subspaces (cf. figure 3.5). The output of the network is a feature map, where each channel contains data for one classified primitive. To have sufficient data for training the network, they established the floor plan dataset ZSCVFP containing 10,800 colorful 2D floor plan images. Dong et al. claim, that their EdgeGAN solution is faster, "than the conventional and object-detection-framework-based pipeline with minimal performance loss" Dong et al., 2021. The information extracted by the EdgeGAN can be used for creating 3D models out of 2D image floor plans. Dong et al., 2021



Original floor plan Segmentation floor plan Subspace connective graph

Figure 3.5: Vectorization steps based on EdgeGAN including the original floor plan, its segmented version and the subspace connective graph Dong et al., 2021

3.2 Automatic classification of the Level Of Geometry (LOG)

The Level Of Detail (LOD) defines the complexity of the available information model. During the ongoing design process of a building, the model should develop from a coarse design to a detailed model including both geometric and semantic, data associated information.

The Level Of Geometry (LOG) defines the level of detail for the geometric appearance of elements within the model and summarises in combination with the Level Of Information (LOI), which describes the level of invisible, technical, non-geometric information, the LOD.

The LODs play a great role in contracts and execution plans, thus the requirements need to be fulfilled in order to deliver and exchange building models. Also LOD definitions are rather vague and can be interpreted differently by the offices. An automated checking of the LOD is therefore essential in order to avoid inconsistencies. Several BIM tools offer the possibility to automatically check the LOI of the model, but offer no automated review of the LOG.

Abualdenien and Borrmann created an ensemble-learning approach for classifying the LOGs of building elements Abualdenien and Borrmann, 2022. They built a LOG Dataset according to popular LOD specifications and extracted various geometric features from the dataset like the amount and length of edges, faces and vertices in order to extract the elements complexity. Based on the extracted features two tree-based models are trained to classify the buildings elements LOG. Abualdenien and Borrmann, 2022

3.3 Automated floor plan reconstruction from 3D scans

Extracting drawings and linking them back to the 3D BIM model is one challenge. Still, there exist many houses without the existence of a fitting floor plan and accordingly also no digital BIM model. Liu et al. developed a deep learning based solution for creating 2D floor plans out of images taken within a house. They propose a unified framework called FloorNet for automatically reconstructing 2D floor plans from RGBD video scans recorded with normal mobile phones. RGBD videos provide point clouds including both RGB colour information as well as the depth or in other words the relative distance for each point. The floor plan output of the FloorNet is shown in figure 3.6. Liu et al., 2018

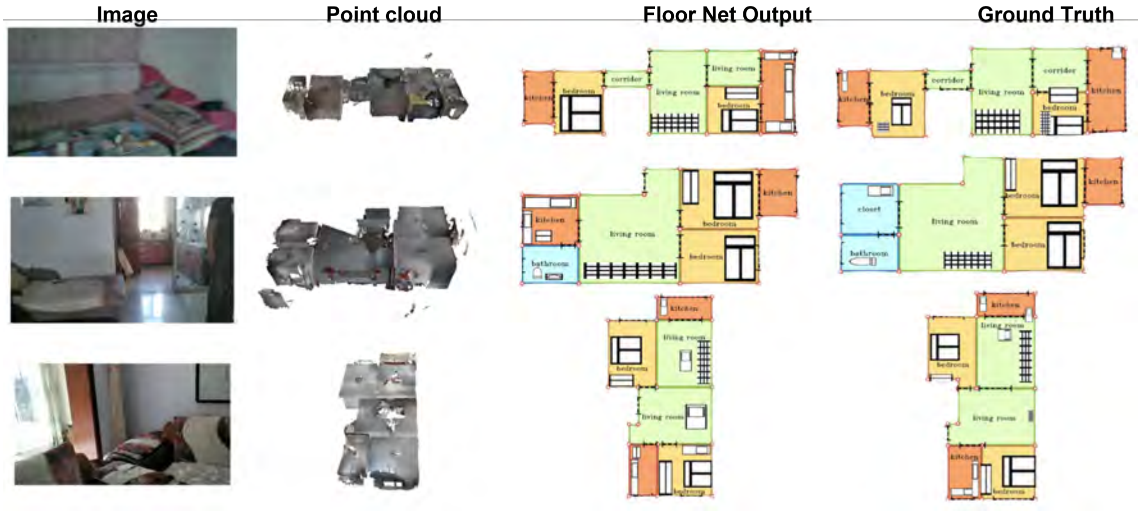


Figure 3.6: FloorNet result of reconstructing floor plans from image scans Liu et al., 2018

The deep learning based architecture FloorNet consists of three branches, that are neural networks exploiting information from different input. The first model retrieves the 3D information from 3D points. The second branch is a CNN, that takes care of the local spatial reconstruction by looking into 2D point density images in a top-down angle. The last CNN processes the full RGB image information. The branches are interconnected for an exchange of feature information and visualised in figure 3.7. Liu et al., 2018

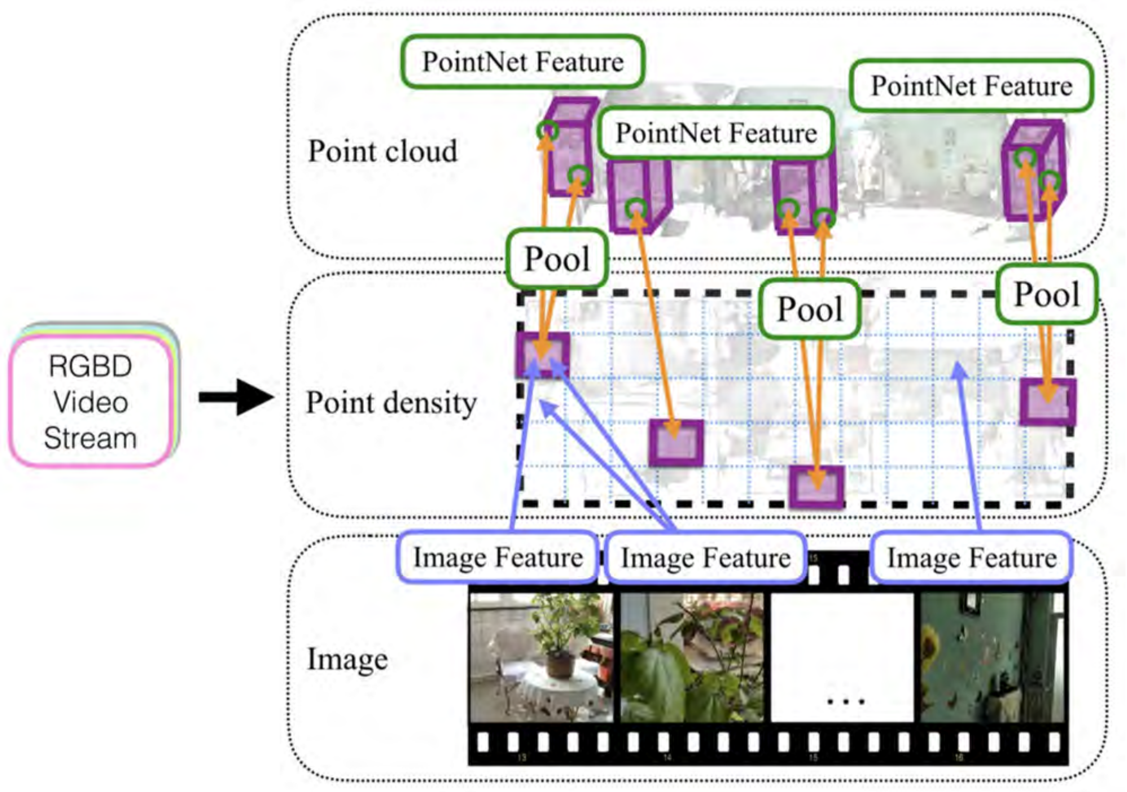


Figure 3.7: Structure of the FloorNet with its three branches Liu et al., 2018

Chapter 4

Concept

The motivation, approach and aim of automatically translating handwritten symbols on Floor Plans (FPs) into pixel based digital symbols using deep learning is described in this chapter. Also, a processing pipeline for automatically integrating handwritten symbols on printed floor plans into the 3D BIM model is proposed.

4.1 Motivation

Nowadays it is possible to derive drawings with various focuses from commercial BIM design applications being at the same status as the BIM model. However, in industry still there are established processes, that refine and develop the building on separate drawings. This leads to the problem of inconsistencies between drawings and the digital model. On the construction site for instance, it is up to now common practice to work with printed drawings.

Thus, floor plans are often altered using handwritten symbols. These handwritten annotations need to be translated manually into the original digital model in the office. The translation process takes time, is an annoying task for the workers and might lead to inconsistencies, clashes or translation errors between the drawing and the model. This workflow is visualised in figure 4.1.

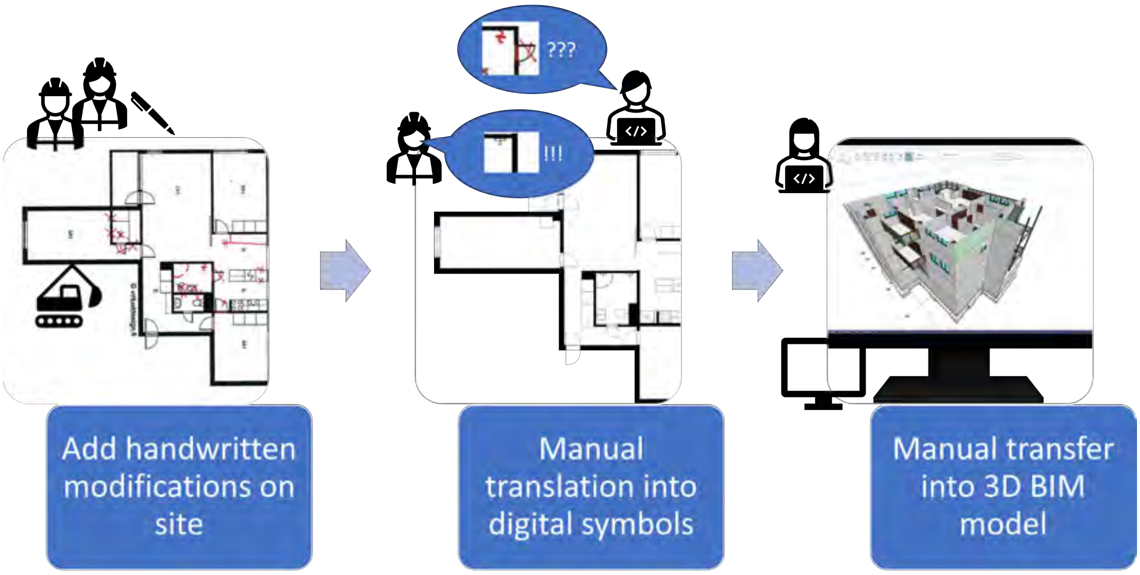
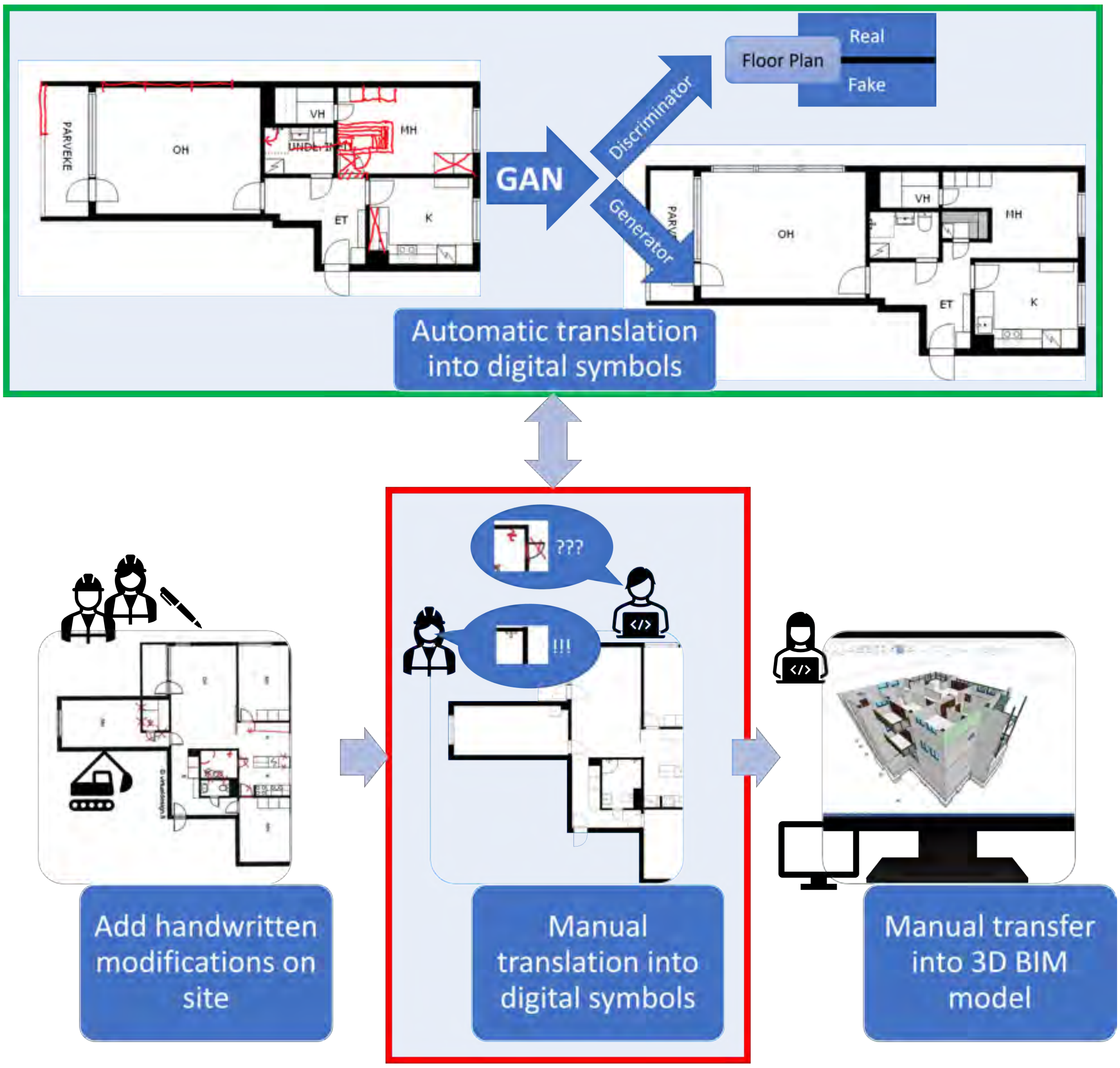

???
!!!
Add handwritten modifications on site
Manual translation into digital symbols
Manual transfer into 3D BIM model

Figure 4.1: Original workflow for transferring handwritten modifications

An automated solution is a deep learning based workflow that transfers the handwritten annotations on a printed drawing into the 3D BIM model and thereby also checks the alterations. This chapter suggests possible steps for such a workflow and also explains the concept of the first step of this workflow to design a prototype deep learning based model for automatically detecting and translating the handwritten symbols on scanned drawings into the pixel of digital symbols. Figure 4.2 shows the embedding of this automatic translation process into the original workflow.



GAN
Discriminator
Generator
Floor Plan
Real
Fake
Automatic translation into digital symbols
???
!!!
Add handwritten modifications on site
Manual translation into digital symbols
Manual transfer into 3D BIM model

Figure 4.2: Automated translation of handwritten modifications into digital symbols embedded into original workflow

4.2 Approach

Within this thesis a prototype GAN-model for translating the pixel of handwritten symbols on floor plans into the pixel of digital symbols is created and evaluated. The aim is to proof, that a deep learning based translation of handwritten symbols on floor plans is possible.

4.2.1 Problem Description

In the context of this thesis handwritten symbols on architectural floor plans describe specific modification actions on geometric features within the floor plan. Some floor plan datasets for various deep learning applications already exist, but none is appropriate for the within this thesis examined translation task. Thus, the new dataset 'H-Symb FP' is created. In chapter 5 related datasets, the creation pipeline of the H-Symb FP dataset, the simplification assumptions and the modification symbols are explained in detail.

The floor plans are born digitally, which means, that the only handwritten signs are modification symbols. To simplify the dataset creation process the handwritten modification symbols on the floor plans are added directly on the digital floor plan using a drawing program and a digital pen. Also, the modifications are drawn always in the same red colour on a born digital greyscale floor plan. Besides the modification symbols, the input and output floor plans are completely identical in shape, size and pixel values.

The task of the model is to automatically identify the manual changes, understand them, in the best case check them for feasibility, translate the handwritten pixel into digital pixel and place the symbol correctly and reasonable within the floor plan image. It should for example be able to understand, that a window in the middle of a room, being not located within a wall, does not make sense and skip this symbol automatically. Figure 4.3 shows the ideal automated process using a GAN model for step 2 in figure 4.1.

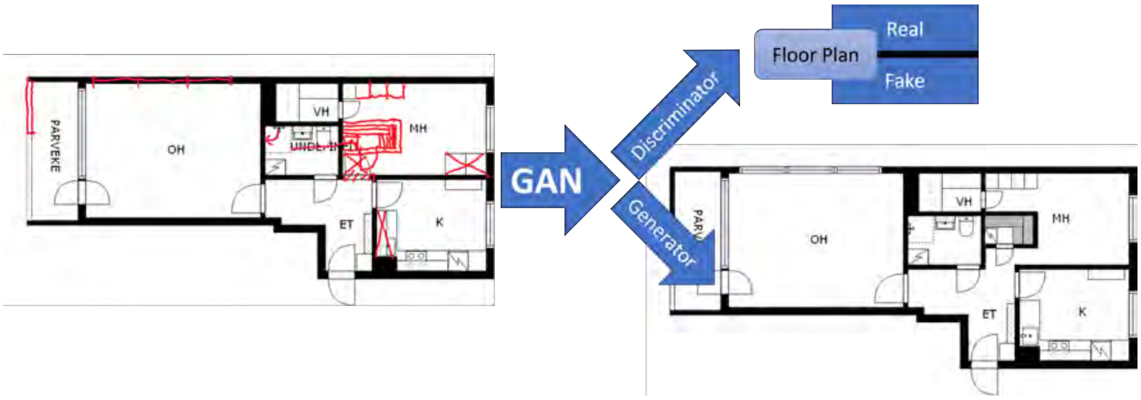


Figure 4.3: Input and output of the GAN model

4.2.2 Format discussion

Drawings are either represented by pixel or vector based descriptors. Depending on the intended purpose of the floor plan both a .png and .svg format can make sense. Each sample in the H-Symb FP dataset contains the input and output images in a pixel based representation as .png file as well as in a vector based representation with a .svg label. Thus, it can be used for training networks with different input and output format combinations.

.png to .png This thesis focuses on training a pixel based GAN to translate an input .png image into another output .png image. Thereby the semantics and classification of the symbols are neglected. This translation helps to avoid misconceptions in the interpretation of the handwritten symbols and provides an easier readability of a manually modified drawing.

.png to .svg When the semantics and classification of geometric features within a floor plan also play an important role, the H-Symb FP dataset can further be used for training a network to predict the .svg format from a pixel based representation. Also, the handwritten symbols are translated into svg elements.

.svg to .svg To train a network to predict from an input .svg an output .svg, the input .svg files in the H-Symb FP dataset first need an alteration. The handwritten symbols in the pixel based representation need to be translated into a vector based description and added into the input.svg file. Therefore, first all red pixel are extracted from the InputWithAnn.png image and converted with common tools into a vector based path representation always keeping the same size and ratio. Secondly, this representation of all handwritten modifications is concatenated with the original input.svg file.

4.2.3 Selection of model architecture

Mapping pixel based input floor plan images to pixel based output floor plan images is a task belonging to the application field of image to image translation introduced in section 2.4.2. Both CNNs as well as GANs can learn the pixel wise image to image translation. A CNN focuses on translating pixel as close as possible to the target image, whereas GANs focus on producing images, that look as real as possible.

The classification of real and fake image parts is ranked by the discriminator, that also penalises the generator in case of fake predictions. The discriminator provides accordingly a completely new evaluation and training possibility of the generator, as the generator learns indirectly from the data. Hoping, that a GAN model is able to capture more or at least other hidden features within the floor plan images compared to a CNN, for example facts like windows and doors can only exist in walls, furniture elements are usually aligned along the walls, kitchen elements do not exist in bathrooms and so on, a combination of a GAN and CNN architecture is chosen for solving the translation task of this thesis.

The 350 floor plan images in the H-Symb FP dataset provide data for training a model, that learns supervised, but form a rather small dataset. GANs learn due to the contest between the Generator and the Discriminator, providing therefore a good basic network structure for the task and a new way of learning to generalise.

Isola et al present a cGAN model being able to translate an input image into another scenery and also claim, that a dataset consisting of 90 to 400 unique images can be enough Isola et al., 2016. This 'pix2pix' network (cf. section 2.4.3) can be trained on many different image to image mapping applications. The generator learns both through penalties by the discriminator as well as through the L1 difference between the predicted and the target image.

Accordingly, the advantages of both the GAN and the CNN structure are combined within this pix2pix network. Expectantly, this combination should both teach the model to predict floor plans as close as possible to the target image as well as to create symbols being as realistic as possible. Thus, that network architecture is taken as basic concept structure within the here developed model. Educated guessing, trial and error, comparing the loss curves of training and validation data and visual evaluation, are used for optimising hyperparameters and the layer architecture.

4.2.4 Aim and expected outcome

The aim of this thesis is to create a FP dataset for analysing the ability of the pix2pix cGAN model to learn supervised from images to translate handwritten annotation symbols within floor plans into digital symbols in the output floor plan. The thesis should prove, that a GAN model is theoretically able to fulfill the given task and outline the limitations.

Also, the performance of the cGAN-prediction is compared with the performance of a similar CNN model architectures prediction, where the loss function contains only the L1-distance. To best knowledge the model parameters should be optimised in order to obtain the best possible results.

When there are enough training samples and variants of a handwritten symbol representation the cGAN model is expected to predict the fitting digital symbol. Also, its ability to generalise should be better than using a pure CNN if enough data is available, as two powerful prediction tools are combined in the pix2pix architecture.

The evaluation is done visually and based on the loss curve of the generator using training and validation data. Based on some test cases, the capabilities, limitations and further future ideas of the trained model are discussed.

4.3 Pipeline for transferring handwritten changes

The following workflow suggests an idea, how to automatically transfer handwritten annotations in analog 2D drawings to the corresponding BIM model. The processing steps within the here proposed pipeline are based on related work introduced in chapter 3. The embedding of this pipeline into the original working practice is illustrated in figure 4.4.

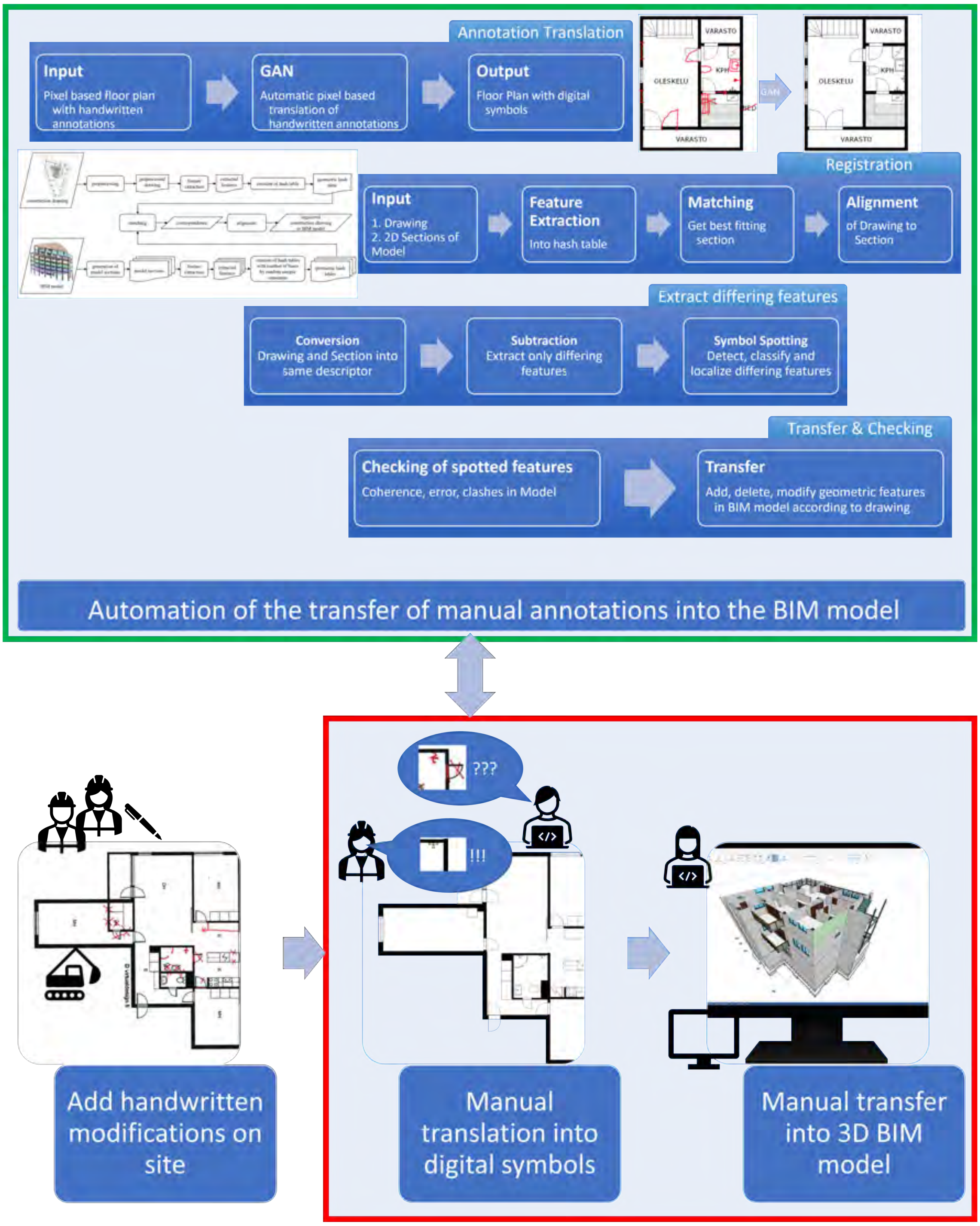

Annotation Translation
Input
Pixel based floor plan with handwritten annotations
GAN
Automatic pixel based translation of handwritten annotations
Output
Floor Plan with digital symbols
VARASTO
OLESKELU
KPH
GAN
Registration
Input
1. Drawing
2. 2D Sections of Model
Feature Extraction
Into hash table
Matching
Get best fitting section
Alignment
of Drawing to Section
Extract differing features
Conversion
Drawing and Section into same descriptor
Subtraction
Extract only differing features
Symbol Spotting
Detect, classify and localize differing features
Transfer & Checking
Checking of spotted features
Coherence, error, clashes in Model
Transfer
Add, delete, modify geometric features in BIM model according to drawing
Automation of the transfer of manual annotations into the BIM model
???
!!!
Add handwritten modifications on site
Manual translation into digital symbols
Manual transfer into 3D BIM model

Figure 4.4: Embedding into original workflow of concept for automated transfer of analog annotations into the digital BIM model

The concept established within this thesis is a processing step for translating handwritten modifications on drawings into a readable digital format. This format can then further be used to automatically transfer the modifications into the 3D BIM model. In the following, the processing steps are described in detail. The corresponding pipeline is visualized in figure 4.5.

Annotation Translation

- As a first step a pixel based representation of the printed floor plan with the handwritten annotation symbols is created by scanning the drawing.
- The within this thesis proposed GAN model is used for translating the handwritten symbols into digital symbols using pixel based descriptors. The output of the model is still a pixel based FP.

Registration

- The next step is the registration of the pixel based digital floor plan to the corresponding 2D section within the BIM model. Using 'extended geometric hashing' described in section 3.1.2 the similarity of the sections is measured and the matching extract from the full 3D BIM model is found TRZECIAK and BORRMANN, 2018.
- To align the drawing and the section with the best similarity score, the section is cut, rotated and scaled to match the shape of the input floor plan. The translation, rotation and scaling vector is stored for later usage.

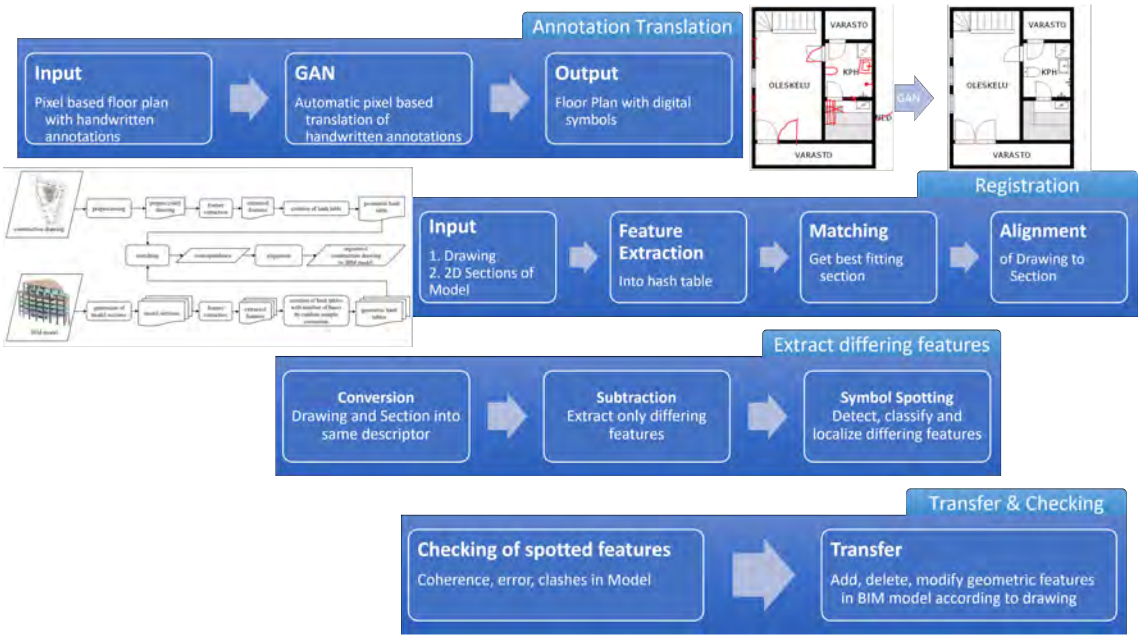
Symbol Spotting of differing features

- The drawing and the section are translated into the same descriptor. Either the pixel based description of the floor plan is translated into a vector based description using common .png to .svg format translators. The EdgeGAN proposed in subsection 3.1.3 can for example provide such a translation. Or, the section is converted into pixel based descriptors and thereby ignoring semantic information.
- Spot the inconsistencies between the drawing and the section by subtracting the pixel in case of pixel based descriptors. In case of vector based descriptors the paths can be compared.
- For detecting, classifying and localizing the differing features various symbol spotting methods can be used depending on the descriptor. The various methods of Symbol Spotting are summarised in section 3.1.1 and explained in more detail in REZVANIFAR et al., 2019.

Checking and Transferring

- Translate the local coordinates of all spotted symbols into the global coordinates of the 3D BIM model. Therefore using the beforehand retrieved translation, rotation and scaling vector.
- Check for coherence, semantic errors, statics and clashes for each modification action.

- Transfer the add, delete and modify annotations into the 3D BIM model according to the drawing using the calculated global coordinates and symbol spotting methods. In case of detected problems, throw an error and propose another fitting variant of the problematic symbol.



Annotation Translation

Input
Pixel based floor plan with handwritten annotations

GAN
Automatic pixel based translation of handwritten annotations

Output
Floor Plan with digital symbols

Registration

Input
1. Drawing
2. 2D Sections of Model

Feature Extraction
Into hash table

Matching
Get best fitting section

Alignment
of Drawing to Section

Extract differing features

Conversion
Drawing and Section into same descriptor

Subtraction
Extract only differing features

Symbol Spotting
Detect, classify and localize differing features

Transfer & Checking

Checking of spotted features
Coherence, error, clashes in Model

Transfer
Add, delete, modify geometric features in BIM model according to drawing

Figure 4.5: Processing Pipeline for automatically transferring handwritten annotations from a drawing into the BIM model

Chapter 5

Floor Plan Dataset

A proper and also extensive dataset is essential for a good performance of a DNN. A dataset for a certain application must be selected, processed or created carefully in order to not fool, overfit or underfit the network. Therefore, the output of the model stands and falls with the dataset.

Within the framework of this thesis an extensive research for online available floor plan datasets is executed in order to find a matching dataset. In the following chapter these online available floor plan datasets are introduced and their applications are discussed, especially with regards to the challenge of finding a deep learning based solution for a pixel wise translation of handwritten symbols within floor plans.

As no appropriate dataset is found, the new floor plan dataset H-Symb FP is created in order to get the best possible performance for the within this thesis proposed deep learning task, described in more detail in chapter 4. This chapter summarises the creation pipeline, structure and purpose of the newly created dataset.

5.1 Available Floor Plan Datasets

Several datasets consisting of both vectorized and pixel based floor plan documents are available online. They were created, modified and labelled in order to train AI models to perform tasks like automatic parsing of floor plan images into another format, document retrieval or semantic segmentation. Table 5.1 summarizes online available floor plan datasets including the amount of unique floor plan samples within the dataset.

Table 5.1: Summary of online available floor plan datasets

Online available floor plan datasets			
Name	Amount	Purpose	Section
CubiCasa5K	5000	extract geometric and semantic information	5.1
Rakuten	500	pixel-wise wall segmentation	5.1
CVC-FP	122	semantic segmentation	5.1
ROBIN	510	floor plan retrieval from sketch	5.1
BRIDGE	13000	symbol spotting, automatic region description	5.1
SESYD	10x100	symbol spotting, document retrieval	5.1
SFPI	10x1000	symbol spotting, document retrieval	5.1
FPLAN-POLY	42x50	symbol spotting	5.1
ZILLOW	2564	3D Scene understanding from 2D images	5.1
FloorPlanCAD	15000	panoptic symbol spotting	5.1

CubiCasa5K

The CubiCasa5K large scale floorplan image dataset was published in 2019 and contains 5000 annotated samples. Over 80 floorplan object categories are distinguished and labelled. Each sample consists of a pixel based image or scan of a floor plan and a vector based representation with SVG label. The svg document consists of various polygons representing the geometry of the various objects. Each object has a unique geometric representation and also a semantic label. Ahti Kalervo et al., 2019; Kalervo et al., 2019

The dataset is separated into floorplan images with colour and greyscale images and also into architectural and high quality greyscale images. The floorplan samples vary in size, amount of objects, rooms, shape and style of the image. Also, there are around 300 floorplans with handwritten annotations, that created the basic idea of this thesis to train a network, that can also read handwritten annotations. In context of this thesis the samples with handwritten annotations were sorted into a new, smaller dataset. The full dataset can be downloaded following the link: https://zenodo.org/record/2613548. Ahti Kalervo et al., 2019; Kalervo et al., 2019

This dataset is intended for finding a solution to the problem, that semantic and geometric information is lost in the process of rasterizing CAD created floor plans into pixel based images, when the floor plans are printed or published. Kalervo et al. present a multi task CNN trained on the CubiCasa5K dataset, that is able to predict the geometry and semantic information of floor-plan images Kalervo et al., 2019. Figure 5.1 shows one sample of the dataset and a possible prediction from the multi task model for floorplan image analysis. Ahti Kalervo et al., 2019; Kalervo et al., 2019

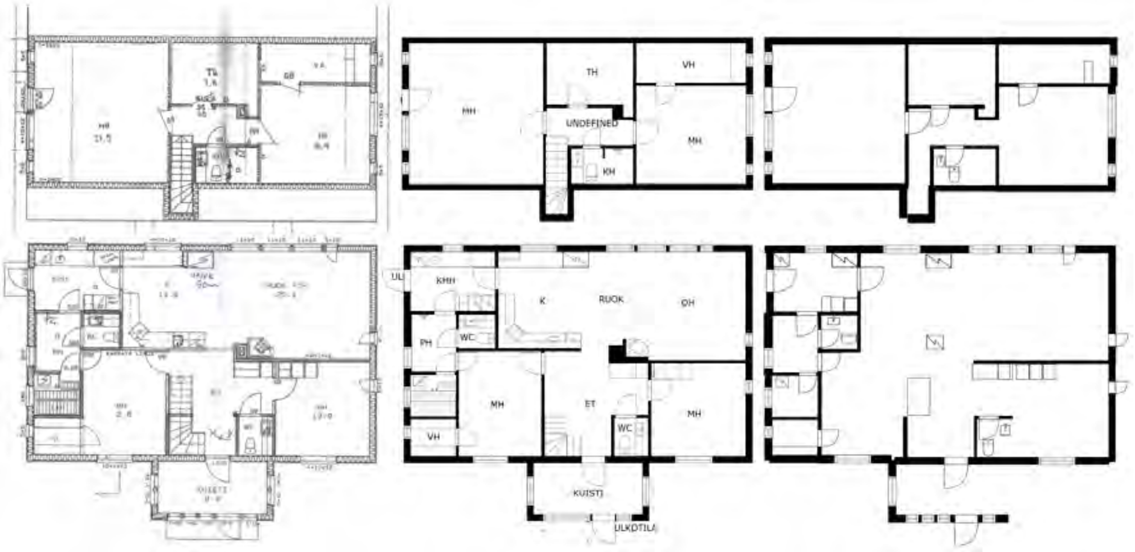


Figure 5.1: Sample from the CubiCasa5K dataset with the original floor plan image on the left, the SVG label in the middle and on the right an automatic prediction Kalervo et al., 2019

Rakuten Dataset

The dataset created by the Rakuten Institute of Technology (RIT) contains 500 real estate floor plans. Each sample includes an image of a real estate floor plan with furniture symbols. Some rooms have size labels. Also, the wall positions are annotated in pixel for each floor plan. One sample is shown in figure 5.2. To get access to the R-FP-500 dataset for research purposes an application form is necessary and an agreement needs to be signed with the RIT. Dodge et al., 2017

Researchers from the RIT trained a fully convolutional network using this dataset in order to segment walls in floor plan images, in other words to detect floor plan wall pixel. Also, optical character recognition is used in order to retrieve the size of the rooms. The main goal of the RIT is to convert floor plan images into a parametric model. Further information can be found in the paper "Parsing floor plan images". Dodge et al., 2017

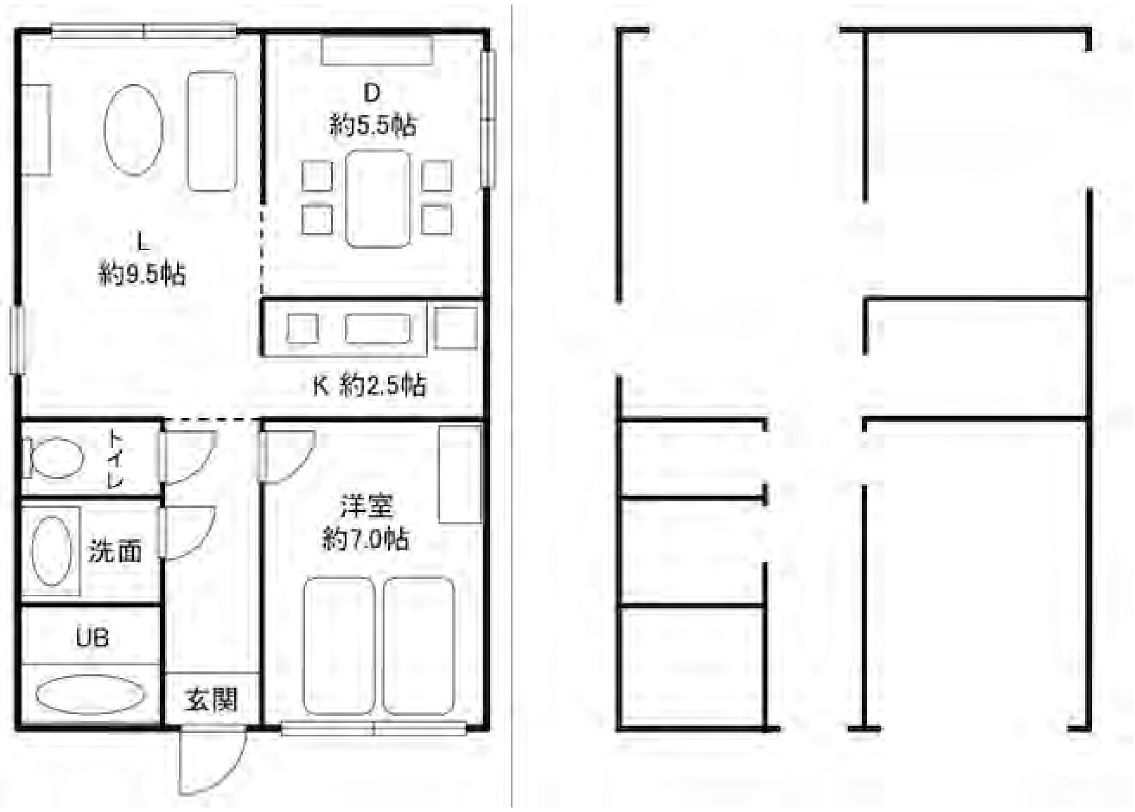


Figure 5.2: Sample from the real estate floor plan dataset by the RIT with the original floor plan image on the left including furniture and room size labels and the pixel wise wall label on the right Rakuten Institute of Technology, 2017

Document Analysis Group (CVC-FP)

The Document Analysis Group published in 2015 a database for structural floor plan analysis, the CVC-FP dataset. It is a collection of 122 scanned floor plan documents of different styles, origin, qualities, resolutions and modeling styles. Each sample is fully groundtruthed and thereby differs between the structural symbols: walls, rooms, doors, windows, parking doors and room separations. There are four different layout classes. Four samples including their ground truth are shown in figure 5.4. The dataset is published and can be downloaded here: http://dag.cvc.uab.es/resources/floorplans/. The intended usage of the dataset are wall segmentation and room detection tasks. de LAS HERAS et al., 2015

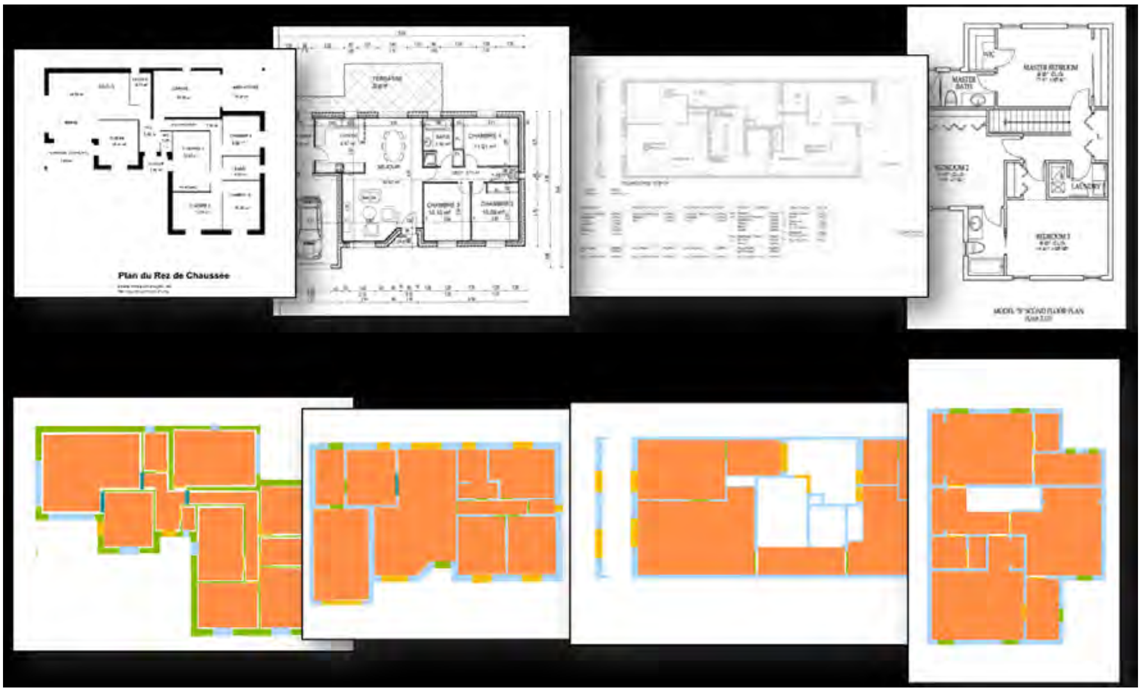


Figure 5.3: Four samples from the CVC-FP dataset with the scanned floor plan document on top and the ground truth below de LAS HERAS et al., 2015

ROBIN dataset

The Repository Of BuildIng plaNs (ROBIN) is a dataset containing binarized images of floor plans with differing arrangements of furniture objects inside the rooms. The dataset is split into three categories each containing 170 images. The categories are divided depending on the amount of rooms within the floor plan; either three, four or five rooms. For each image a self made hand sketch was created. The dataset is used for training a network, that detects similarities and finds the match between the sketch and the actual floor plan image. This is called document retrieval. It can be downloaded here: https://github.com/gesstalt/ROBIN. D. SHARMA et al., 2017

BRIDGE

The Building plan Repository for Image Description Generation, and Evaluation (BRIDGE) dataset consists of more than 13000 images of floor plans. Each floor plan has symbol annotations, region and paragraph descriptions, see figure 5.5. Thus, the dataset can be used for symbol spotting tasks, caption generation and the automatic description of

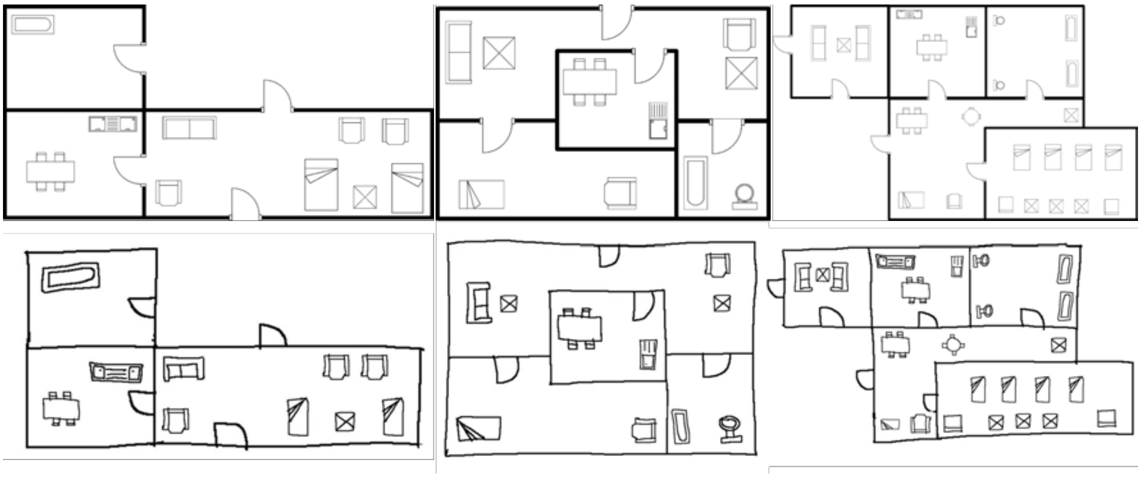
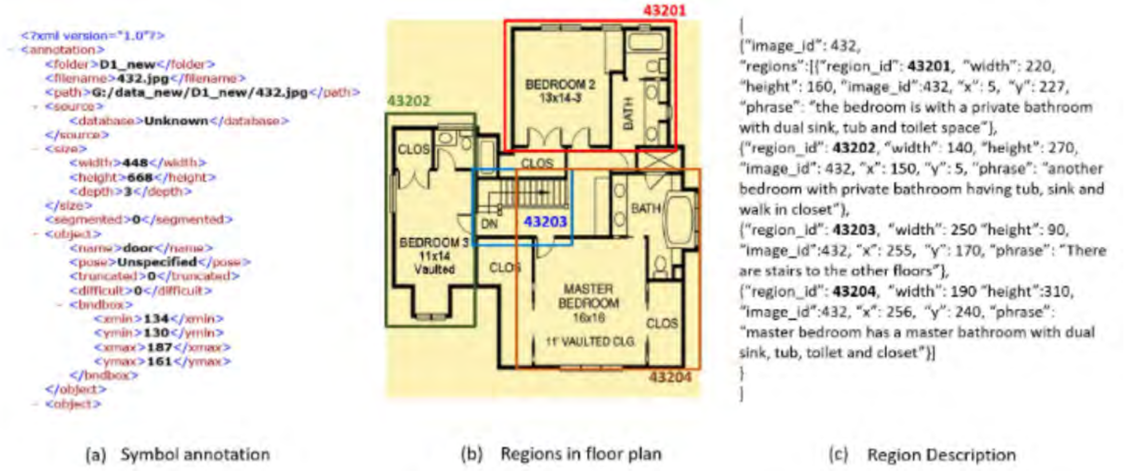


Figure 5.4: Samples from each category of the ROBIN dataset with the floor plan image on top and the matching sketch below D. SHARMA et al., 2017

sections of the floor plan. The dataset can be downloaded from github following the link: https://github.com/gesstalt/BRIDGE. GOYAL et al., 2019



```
<?xml version="1.0"?>
- <annotation>
    <folder>D1_new</folder>
    <filename>432.jpg</filename>
    <path>G:/data_new/D1_new/432.jpg</path>
  - <source>
      <database>Unknown</database>
    </source>
  - <size>
      <width>448</width>
      <height>668</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>door</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>134</xmin>
        <ymin>130</ymin>
        <xmax>187</xmax>
        <ymax>161</ymax>
      </bndbox>
    </object>
  - <object>
```

(a) Symbol annotation

(b) Regions in floor plan

```
[
{"image_id": 432,
"regions":[{"region_id": 43201, "width": 220,
"height": 160, "image_id":432, "x": 5, "y": 227,
"phrase": "the bedroom is with a private bathroom
with dual sink, tub and toilet space"},
{"region_id": 43202, "width": 140, "height": 270,
"image_id": 432, "x": 150, "y": 5, "phrase": "another
bedroom with private bathroom having tub, sink and
walk in closet"},
{"region_id": 43203, "width": 250 "height": 90,
"image_id":432, "x": 255, "y": 170, "phrase": "There
are stairs to the other floors"},
{"region_id": 43204, "width": 190 "height":310,
"image_id":432, "x": 256, "y": 240, "phrase":
"master bedroom has a master bathroom with dual
sink, tub, toilet and closet"}]
}
]
```

(c) Region Description

Figure 5.5: Sample from BRIDGE dataset with symbol annotations, the full floor plan and region descriptions GOYAL et al., 2019

SESYD

The System Evaluation SYnthetic Document (SESYD) dataset contains 10 different floor plan images. Each floor plan image is stored 100 times with each sample containing a different arrangement of furniture symbols within the rooms and having different layouts. Each sample has a unique SVG label including semantic and geometric information as well as bounding boxes for the symbols. Figure 5.6 shows one sample with bounding boxes and class labels. The dataset is available here: http://mathieu.delalandre.free.fr/projects/sesyd/. It is used for symbol spotting purposes and retrieval tasks. DELALANDRE, 2014

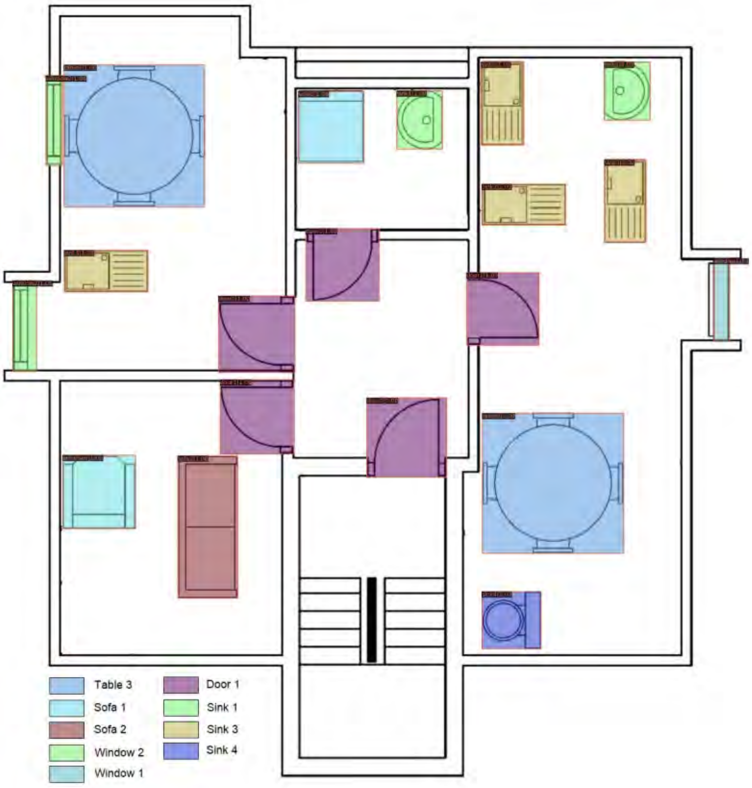


Figure 5.6: Sample from SESYD dataset with ground truth and highlighted furniture class objects MISHRA et al., 2021

SFPI - Synthetic Floor Plan Images

This dataset is based on the basic layouts and object classes of the SESYD dataset floor plans and is also intended for the same usage of symbol spotting and object detection. It contains 10,000 images, including also 10 different floor plan layouts with 16 different furniture class objects and can be downloaded here: https://cloud.dfki.de/owncloud/index.php/s/mkg5HBBntRbNo8X. Each image has another furniture symbol arrangement. In contrast to the SESYD dataset, also object augmentation like rotating and scaling the furniture symbols has been applied, so that the model can generalize better. MISHRA et al., 2021

FPLAN-POLY

FPLAN-POLY is a set of 42 real floorplans, that were vectorized, given both in DWG and PDF format. There are 38 different symbol classes. Further, the set of symbols is given, which can be taken as model queries. The dataset can be used for symbol spotting tasks. At the time this thesis was written, the dataset was not available online anymore. GOYAL et al., 2019

ZILLOW-Indoor dataset

The Zillow Indoor dataset contains 71,474 annotated 360° panoramas, 21,596 room layouts and 2,564 floor plans of real, unfurnished residential homes, shown in figure 5.7. It can be downloaded here: https://github.com/zillow/zind. This dataset can be used for the reconstruction of floor plans from panoramas and also for floor plan analysis. As this dataset is not really fitting into the frame and goal of this thesis, it is not discussed further. CRUZ et al., 2021

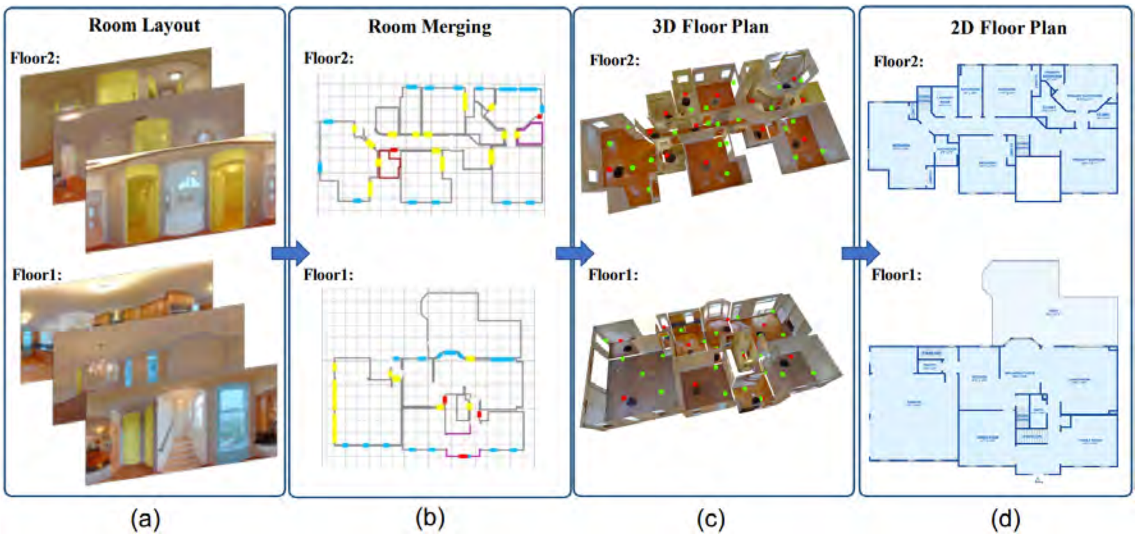


Figure 5.7: Two samples from the Zillow dataset with the panorama room layout images, the merged rooms, the 3D floor plan scan and the final 2D floor plan CRUZ et al., 2021

FloorPlanCAD

The FloorPlanCAD dataset is a large scale and real world CAD drawing dataset containing 15,000 floor plans derived from various residential and commercial building projects. The floor plans are given within the dataset in the svg format providing both semantic and geometric information and also as a png image. The dataset is designed for panoptic symbol spotting tasks, which means, that both instances of countable things and semantics of uncountable stuff are spotted. It can be downloaded here: https://floorplancad.github.io/. FAN et al., 2021

5.2 H-Symb FP Dataset

The H-Symb FP (Handwritten Symbol Floor Plan) Dataset was created in context of this thesis for the purpose of a pixel wise translation of handwritten modifications on floor plan images into the correctly scaled and rotated standard digital symbols embedded in the original image. The 350 samples are derived from the svg label of floor plans from the CubiCasa5K dataset.

5.2.1 Creation pipeline

After a research about available floor plan datasets in the internet, the svg labelled floor plans from the CubiCasa5K dataset were most appropriate for the given task. To create one sample for the H-Symb FP Dataset the following steps were processed, see also figure 5.8:

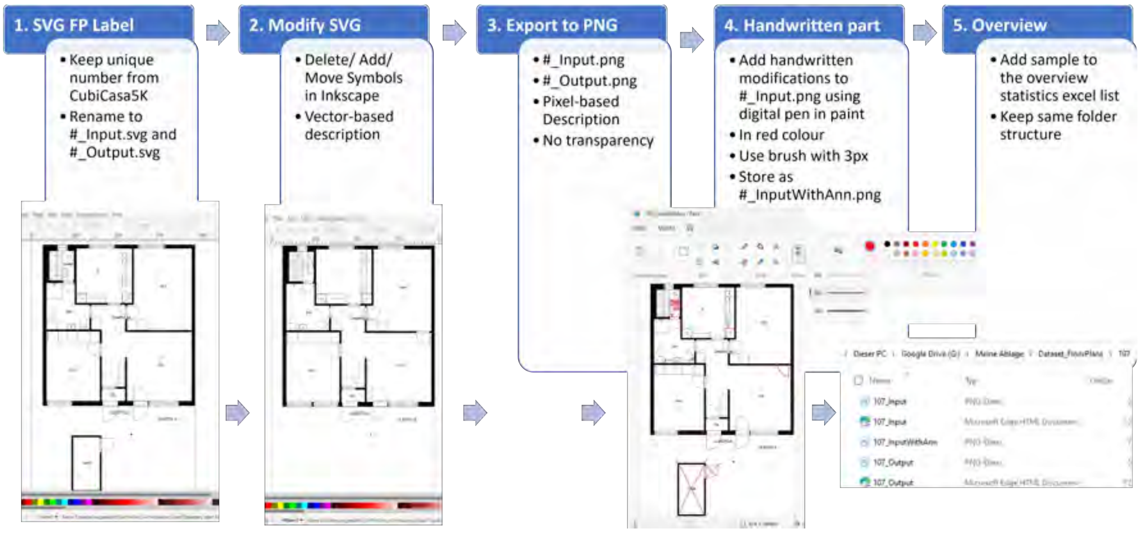

1. SVG FP Label
• Keep unique number from CubiCasa5K
• Rename to #_Input.svg and #_Output.svg
2. Modify SVG
• Delete/ Add/ Move Symbols in Inkscape
• Vector-based description
3. Export to PNG
• #_Input.png
• #_Output.png
• Pixel-based Description
• No transparency
4. Handwritten part
• Add handwritten modifications to #_Input.png using digital pen in paint
• In red colour
• Use brush with 3px
• Store as #_InputWithAnn.png
5. Overview
• Add sample to the overview statistics excel list
• Keep same folder structure

Figure 5.8: Processing pipeline for creating a sample for the H-Symb FP dataset

Processing Pipeline for creating one FP sample:

1. Each FP sample in the CubiCasa5K dataset has an unique number. When a new sample for the H-Symb FP dataset is created, first a folder denominated with this unique number is added in the H-Symb FP dataset. Then the according svg label FP document from the CubiCasa5K dataset is copied into this folder twice and labelled with '{uniqueNumber}_Input.svg' and '{uniqueNumber}_Output.svg' in order to have two FPs being exactly the same.

2. The '{uniqueNumber}_Output.svg' document is edited using Inkscape, which is a free and open source editor for vector graphics documents Inkscape Project, 2022. Furniture element symbols, other basic floor plan shapes and features are deleted, added or modified. As this dataset is intended for pixel based image usage only, the semantic structure was not checked further. This would have taken much more creation time. The modified vector based description of the output is always stored in the .svg format as well.

3. Inkscape also provides the opportunity to export the .svg format into a pixel based representation. Accordingly, the floor plan images '{uniqueNumber}_Input.png' and '{uniqueNumber}_Output.png' are stored as well in the sample folder. Besides the three RGB channel, Inkscape exports a fourth channel for the transparency. By opening both images in paint and storing them again under the same name, the transparency is replaced with white pixel.

4. '{uniqueNumber}_Input.png' is processed further in the Paint software by Microsoft, which is a raster graphics editor also with several drawing functionalities. Using a digital pen and a touch screen, the handwritten symbols matching the modifications done previously in Inkscape are added to the floor plan image. The points discussed in paragraph 5.2.1 are kept in mind when drawing digitally. The brush-functionality with a line thickness of 3px in the predefined red colour (RGB-value$= (237, 28, 36)$, colour number: ED1C24) is always used. The FP image is afterwards stored as '{uniqueNumber}_InputWithAnn.png' in the sample folder.

5. As last step the amount and types of modifications within this newly created sample are added to the Excel Overview sheet - 'ModificationSummary.xlsx'. Each sample can be identified in the overview sheet by the unique CubiCasa5K number.

Mimicking real handwritten annotations The process of printing the floor plan image, sketching the modifications with a pen on the printed image and afterwards scanning the FP again in order to get the digital image version would have been extremely time consuming and a waste of paper. Further, resizing the scanned FP image to exactly the same shape and amount of pixel as the original digital image would be tough. To still keep the digital handwritten modifications as close to real handwritten annotations with a pen on a printed version, the following points were ignored on purpose during the drawing process in paint.

- People sketch their handwritten symbols in a slightly different manner each time they add handwritten modifications to printed floor plans. Thus, the sketched lines were kept as they were drawn without being artificially straightened in order to mimic the natural, slightly zig-zagging pattern in normal handwriting.

- Further, the handwritten symbols are uniquely sketched in every FP again, so that the pixel arrangement, scale and rotation differ slightly, whenever the symbols are used. This creates a natural data augmentation, which helps the model to generalize better and therefore to be able to also recognise the shape of real handwritten symbols better.

- Additionally, the location and size of the handwritten symbols and the matching digital symbol in the FPs are deliberately not exactly the same, but rather sketched using eye estimation.

Selection of Symbol shapes In order to use as familiar and accepted handwritten symbols as possible, another online research for finding the most common handwritten symbol-equivalent to a digital symbol is conducted. The aim is to only apply symbol shapes in the dataset, that are also typically utilised in real world floor plan examples. The final symbol shapes were compared with handwritten annotations found in both the real world floor plan samples from the CubiCasa5K dataset as well as in other online available lists or floor plans. For every symbol class within the H-Symb FP dataset, a similar real world annotation symbol and the corresponding symbol shape used within the H-Symb FP dataset is shown in the tables 5.9 (for all delete symbols), 5.10 (for all add symbols) and 5.11 (for all modify symbols).

5.2.2 Dataset statistics

The H-Symb FP dataset contains 350 Samples with 39 modification classes.

Available modifications

The modification categories are delete, add and modify operations. The handwritten symbol shape for each categorical class were obtained by real world annotations in floor plans e.g. from the CubiCasa5K dataset. The tables 5.9, 5.10 and 5.11 show all modification classes, their occurrences within the dataset and the handwritten symbols used in reality for these annotations (source: CubiCasa5K dataset) as well as their corresponding handwritten symbol within this dataset.

Modification Distribution

The H-Symb FP dataset has 2555 handwritten symbol annotations distributed over all FP samples. Each FP can have different annotation combinations. Figure 5.12 shows the distribution of the amount of annotation symbols per floor plan.

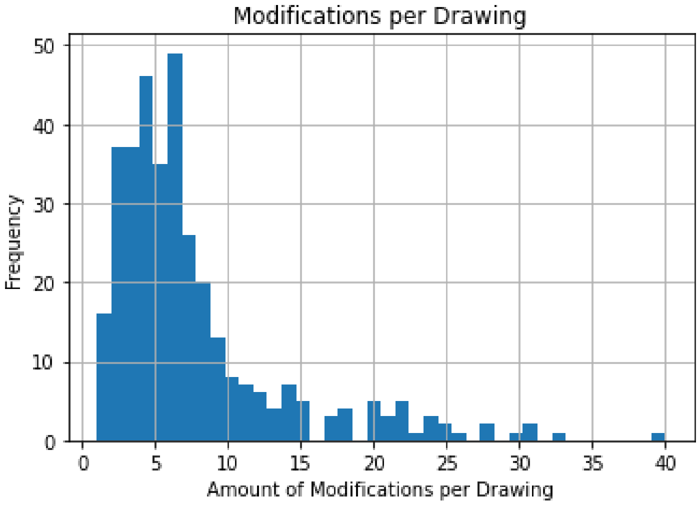


Figure 5.12: Distribution of the amount of handwritten annotations per floor plan

Figure 5.16 compares graphically the class occurrences of each category.

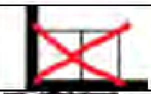
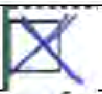
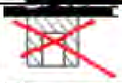
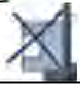
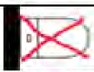
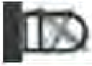
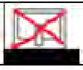
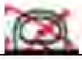
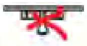
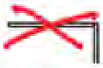
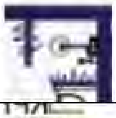
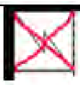
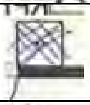
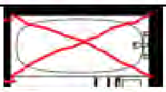
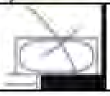
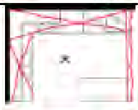
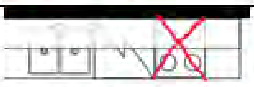
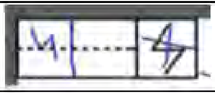
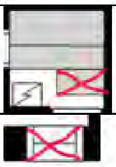
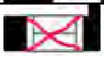
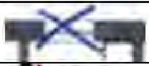
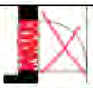
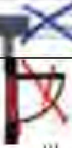
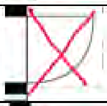
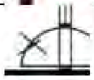
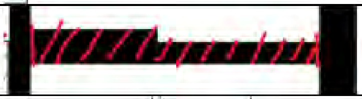
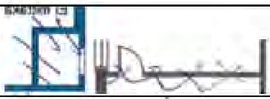
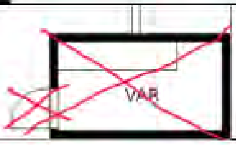
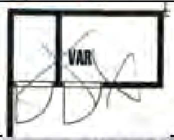
Delete Operation	Handwritten Symbol – H-Symb FP	Symbol in Reality – CubiCasa5K	Occurrence
Cupboard			110
Chimney			27
Toilet			32
Sink			29
Shower			24
Shower Wall			16
Washing Machine			22
Bath Tube			7
Full Kitchen Line			17
Single Kitchen Element			70
Sauna element			35
Window			98
Door & Gap			138
Door			47
Wall			188
Room			54
Room Label	~~UNDEFINED~~	LASIKUISTI ~~AVARAUS~~	79

Figure 5.9: Handwritten symbol classes used for delete operations

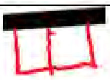
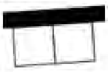
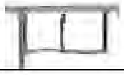
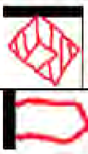
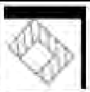
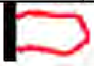
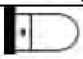
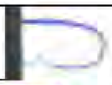
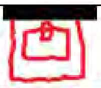
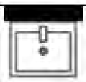
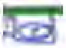
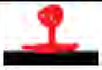
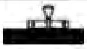
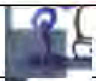
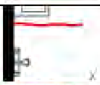
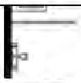
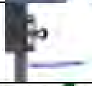
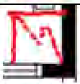
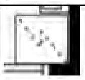
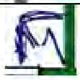
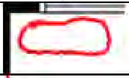
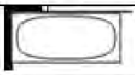
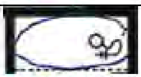
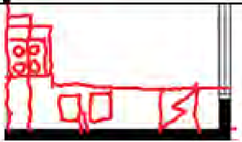
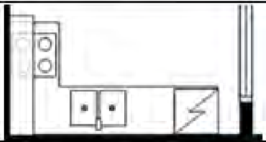
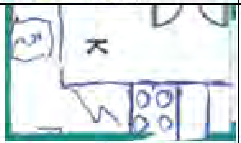
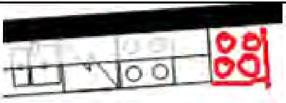
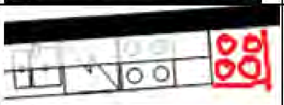
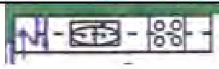
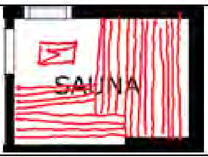
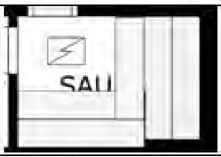
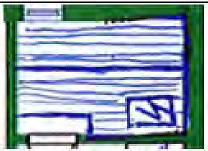
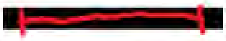
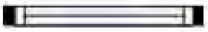
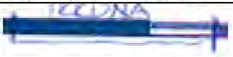
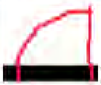
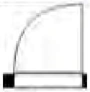
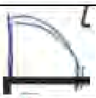
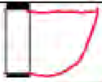
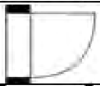
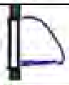
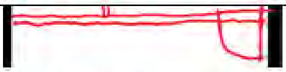
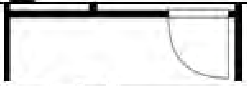
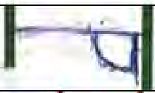
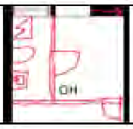
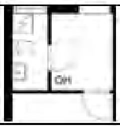
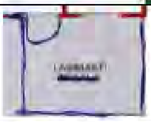
Add Description	Handwritten Modification Symbol	Digital Symbol	Real symbol from Construction Site	Occurrence
Cupboard				68
Chimney				23
Toilet				65
Sink				57
Shower				66
Shower Wall				41
Washing Machine				23
Bath Tube				59
Full Kitchen Line				33
Single Kitchen Element				68
Sauna element				47
Window				192
Door & Gap				157
Door				27
Wall				168
Room				37

Figure 5.10: Handwritten symbol classes used for add operations

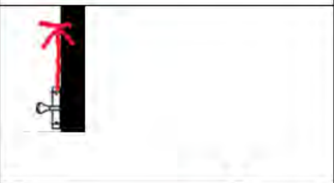
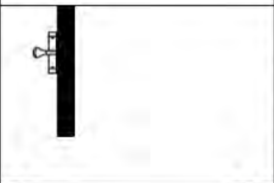
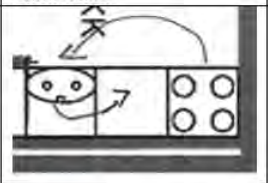
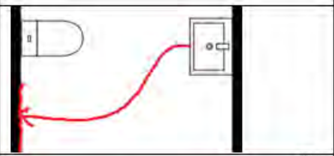
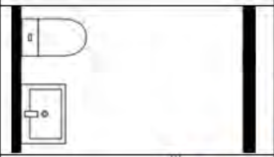
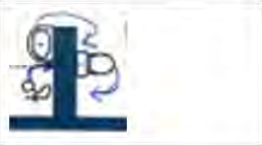
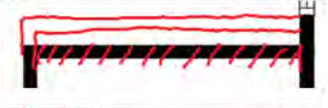
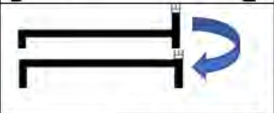
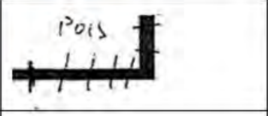
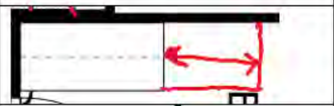
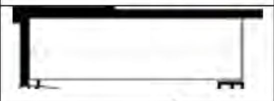
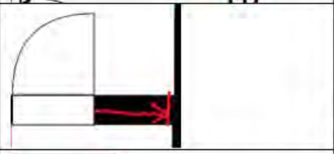
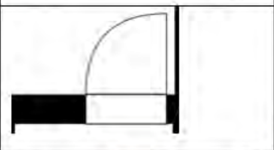
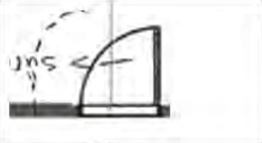
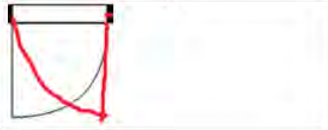
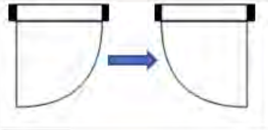
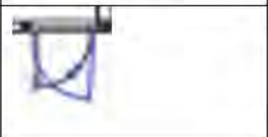
Delete Description	handwritten Symbol – H-Symb FP	Digital Result	Real Symbol from CubiKasa5K	Occurrence
Move Element				100
Rotate Element				77
Modify Wall Shape				43
Modify Element length				82
Move Door & Gap				64
Swap Door Opening				65

Figure 5.11: Handwritten symbol classes used for all feature element modifications

Size of floor plan images

The original floor plans all have different shapes, sizes and width/height ratios. Consequently the ratio of the line thickness to the amount of pixel within the image is not the same in all floor plans. In order to keep all lines after resizing, the interpolation methods needs to be chosen carefully when resizing all floor plans to the same size. Figure 5.17 shows the statistics of the size distribution within the dataset. On the left there is a plot illustrating the frequency of floor plans having a certain amount of pixel as their longest side and the plot on the right displays the distribution of all floor plans having a similar width to height ratio.

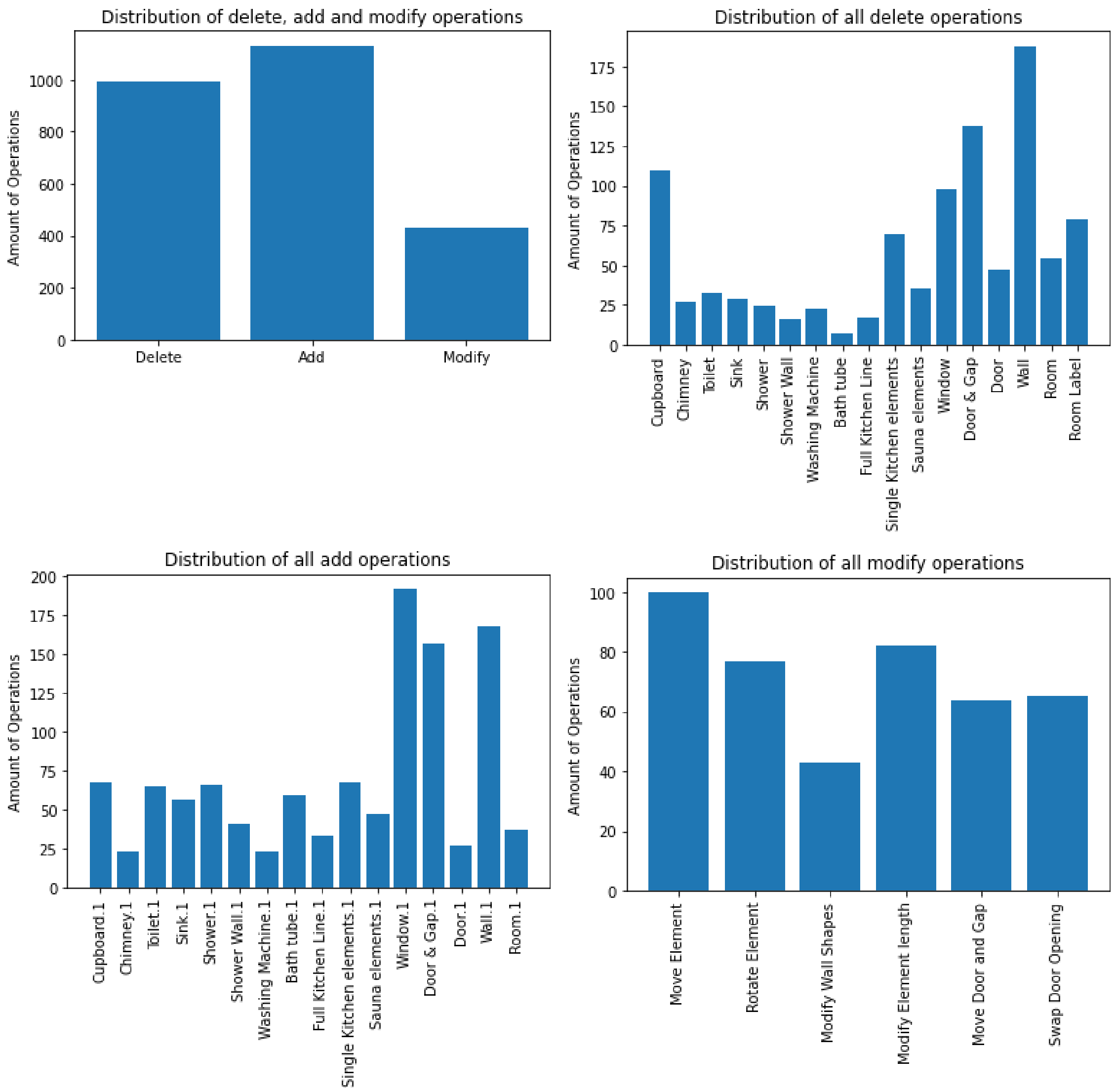


Figure 5.16: Distribution of class instances in the delete, add and modify categories

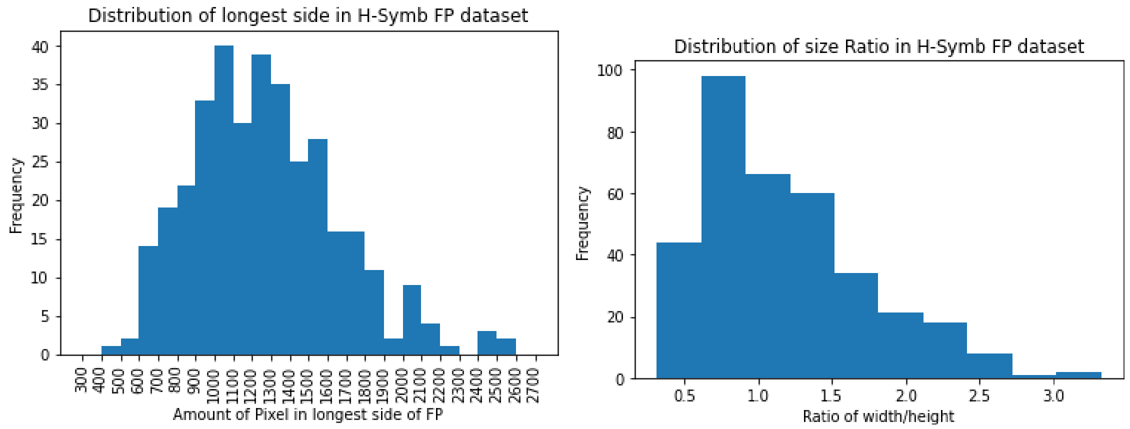


Figure 5.17: Distribution of the longest side (1) and ratio of width/height (2) over all floor plan images in the dataset

Chapter 6

Implementation

In the following, the methods, parameter, algorithms and structure for implementing the cGAN model as well as the preprocessing steps for the dataset are explained in detail. The model is created using Tensorflow and Keras functionalities in python. Tensorflow is an open source library for deep learning tasks and is suitable for handling dataflow programming tasks MARTIN ABADI et al., 2015. Keras also provides an open source neural network Application Programming Interface (API) for designing fast deep neural networks and has been integrated into Tensorflow CHOLLET et al., 2015.

The code for training the model is written in a jupyter notebook environment and executed externally using Google Colab Pro. Google Colaboratory is a research project especially suited for developing machine learning models and provides powerful hardware options like GPU and Tensor Processing Unit (TPU) and also a limited Random Access Memory (RAM) capacity. The model is trained using the available capacity provided by Colab Pro which is 25.46 GB RAM and a NVIDIA P100 GPU architecture.

Unfortunately there is a limited runtime on the servers leading to interrupted execution due to time outs, when there is no activity within the script. When training the model for 1000 steps with a batch size of 1 and an image size of 256x256 it takes the model around 130 seconds to train. Accordingly for one epoch training takes around 400 seconds, when the augmented data is stored as additional samples within the dataset.

6.1 Pre-Processing of dataset

In order to prepare the dataset samples as best as possible for the given task, the following processing steps summarized in figure 6.1 were applied to each floor plan image.

6.1.1 Loading and decoding

The input and output floor plans are stored as .png images within the dataset. As a first step, the image files are read and afterwards decoded into a three channel tensor to reflect an RGB image. The conversion into three channels is important as the images are derived from the .svg format and contain originally four channel. This decreases the model complexity. The values within the tensor are stored as uint8 values. Algorithm 6.1 shows this implementation.

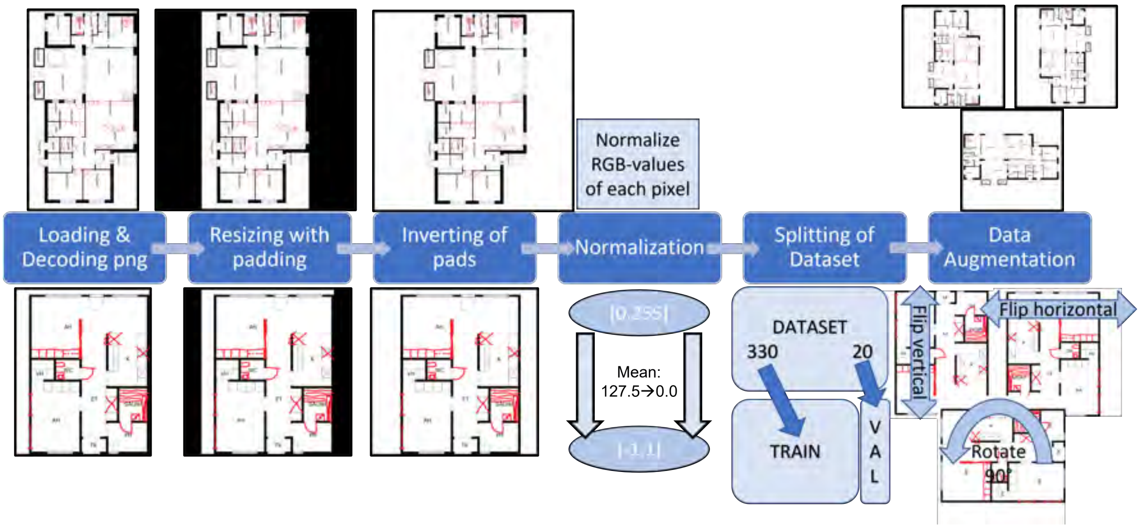


Figure 6.1: Preprocessing steps for all floor plan images in the H-Symb FP dataset

Algorithm 6.1: Loading and Decoding of .png images using tensorflow

```
image = tf.io.read_file(image_path)
image = tf.io.decode_png(image, channels = 3)
```

6.1.2 Resizing

As every floor plan within the dataset is derived from different buildings or at least taken from different building stories, they differ a lot in size and length/width ratio. Also the shape of the story, the amount of rooms and the existing geometric features as well as handwritten annotation symbols vary greatly from one sample to the other. The cGAN model can only process images with the same amount of pixel. Thus, as a first preprocessing step, each Floor Plan is resized to a quadratic shape with a predefined length keeping the original ratio between width and length.

Size A fitting size is a key aspect for training the network efficiently. If especially larger floor plan samples are scaled down too much, thin feature lines and also the handwritten annotations can disappear completely or appear as dashed lines. This confuses the network and leads to false predictions. When however the size of the input images is too big, the computation time for training the network and consumed memory goes up very fast.

Thus, a good trade-off regarding the size is necessary. It should be big enough to still show the pixel of all elements in closed loops but also small enough to not exceed the computation and memory capacities. The sizes of 256x256, 512x512 and 1024x1024 pixel are investigated and compared in terms of computation time, memory consumption and visual appearance. Colab Pro is able to handle sizes of 256x256 and 512x512 relatively well without breaking. Also after resizing, still the information is not lost in both images.

Interpolation method Tensorflow provides a function for resizing the images within a dataset to a custom length and width. It comes with the possibility to choose between different interpolation methods for up- and downsizing. To overcome the problem of erasing lines during downsizing all methods are tested on the largest and smallest image and the output images are compared visually. The area interpolation produces the best downsizing results with still a good visibility of the handwritten annotations, even though the downsizing is more than 0.5 on the largest image. In the appendix in figure A.1 all interpolation methods for resizing the biggest image to a size of 256x256 are compared.

Padding Distortions or random cutting due to up- or downsizing of the floor plans is unwanted. It brings the problem, that the feature elements, wall shapes and symbols might change so significantly, that the model cannot recognise them anymore and they loose important, unique geometrical properties, that are essential for the network to translate the pixel properly. Thus, the original ratio between width and size is kept during resizing.

Still, every image needs to have a quadratic size, even though the original shape is not quadratic. To fulfill this demand, the images are resized with padding, which means, that white pixel are added in order to get a quadratic image. Tensorflow provides only a resizing function, where black pixel are added as pads. To get white pads, the image tensors are inverted bitwise prior to the resizing operation and inverted back after resizing. After the inversion backwards, the uint8 values within the image tensors are cast into float32 values in order to provide continuous input and output values for the network (cf. Algorithm 6.2). Weights, node values and parameters within the network are expected to be continuous instead of being discrete.

Algorithm 6.2: Resizing with white pads using tensorflow

```
image = tf.bitwise.invert(image)
image = tf.image.resize_with_pad(image,TRAINING_SIZE,TRAINING_SIZE,
    method = 'area')
image = tf.cast(image, tf.uint8)
image = tf.bitwise.invert(image)
image = tf.cast(image, tf.float32)
```

6.1.3 Normalization of pixel values

The next step is to normalize all pixels from the standard RGB representation values [0,255] to [-1,1] and move the new mean to zero (cf. Algorithm 6.3). Every pixel is divided by the mean of a pixels maximum value ($= \frac{255}{2} = 127.5$) in order to center the data and get a standard distribution. This helps the network to converge faster, as there are less outliers and also reduces the computation effort.

Algorithm 6.3: Normalizing an image tensor from [0,255] to [-1,1]

```
image = (image / 127.5) - 1
```

6.1.4 Data Augmentation

To increase the amount of different samples within the dataset the input and output floor plan samples are augmented using the two operations flip and rotate. Flipping provides a flip on either a horizontal or a vertical axis and can be compared to mirroring the pixel. The rotation is applied for 90°, 180° and 270° anticlockwise. Other angles are avoided to keep the wall lines within the images in a vertical or horizontal orientation and also to avoid random cutting of the edges, geometric features and handwritten annotations.

Also, the combination of flipping left-right plus rotate 270° as well as flipping upside-down plus rotate 270° were applied in order to get all possible augmentation combinations and still get individual image representations. All augmentation operations are visualized in figure 6.2. Algorithm 6.4 provides the tensorflow integrated functions for flipping left-right and upside-down as well as the rotation by 90° where 'k' defines the amount of rotations. E.g. 'k=3' applies an anticlockwise rotation of 270°. After the augmentation steps the dataset contains 7 times more data than the original dataset.

Algorithm 6.4: Flip and rotate data augmentations

```
image = tf.image.flip_left_right(image)
image = tf.image.flip_up_down(image)
image = tf.image.rot90(image,k=1)
```

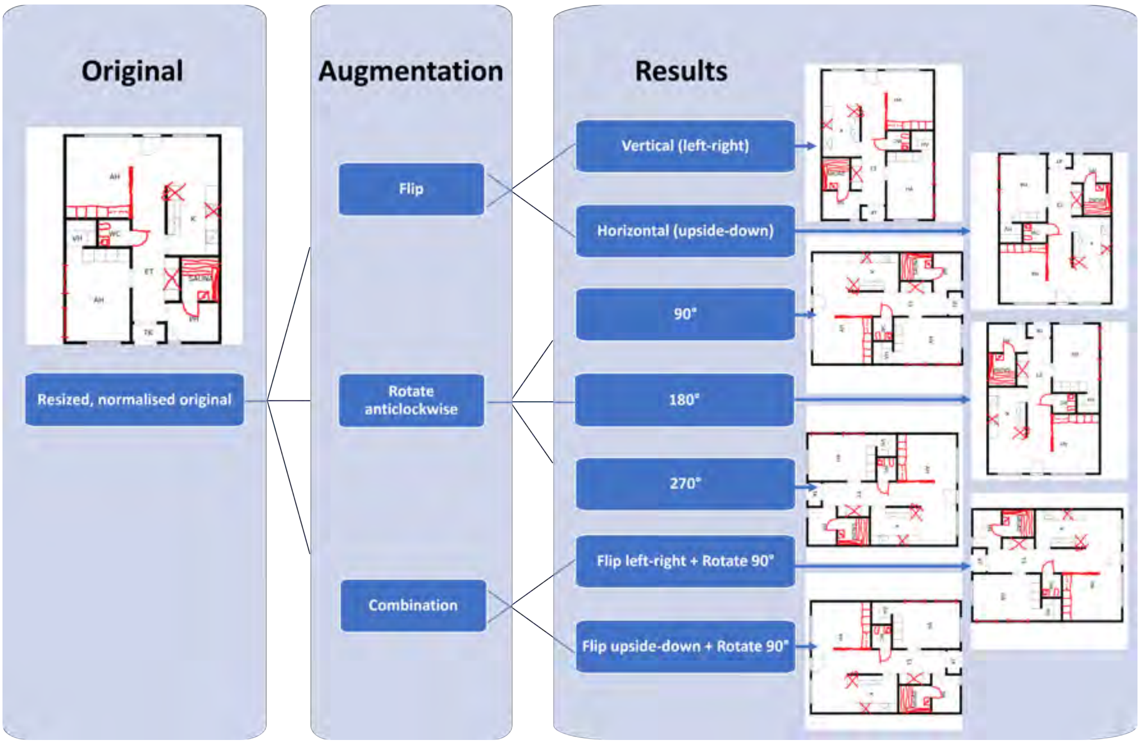


Figure 6.2: All flip and rotate augmentations and combinations used on the dataset

6.1.5 Split into Train, Test and Validation dataset

The first 20 samples are removed from the original H-Symb FP dataset before augmentation and used as validation set. The validation samples are not shown for training to the model. The rest of the samples is augmented and builds the training set comprising 2640 samples. For testing the final network, independently 4 new samples are created including all modification symbols as test set.

Algorithm 6.5: Splitting of tensorflow dataset

```
val_dataset = dataset.take(SPLIT)
train_dataset = dataset.skip(SPLIT)
```

6.2 Selection of Hyperparameters

Hyperparameters have a great influence on the performance of the cGAN model. Small changes can influence the prediction of the model greatly. Also, the runtime and memory needs to be taken into consideration in order to have a reasonable training time. The hyperparameters are chosen via educated guessing and trial and error. Relying on related projects suitable values for the parameters are selected and optimised. Table 6.1 shows the hyperparameters, their initial values and all attempted values. Table A.1 in the appendix shows the training parameter combinations of all training runs.

Table 6.1: Summary of tested hyperparameters for the GAN model

Hyperparameter optimisation for training the cGAN model			
Parameter Name	Initial value	Best value	Tested values
BATCH_SIZE	1	1	1; 2; 4; 8; 16
TRAINING_SIZE	256	512	256; 512; 1024
STEPS	50,000	???	5000 to 350,000
Epochs	19	???	1 to 200
Learning Rate	2e-4	1e-4	1e-3; 2e-4; 1e-4; 1e-5
β_1	0.5	0.999	0.5; 0.999
dataset_size	350	350	plus augmentation
Validation_size	10	20	10; 20
BUFFER_SIZE	2720	2640	Training size for full shuffling
Lambda	100	1000	10; 100; 1000; 10,000; no disc.

Dataset handling The H-Symb FP dataset contains 350 unique floor plans. 20 of these plans are used for validation. The remaining 330 floor plans are augmented in 7 different ways, as the model can generalize better with more data. Thus, the training dataset size is $330 * 8 = 2640$ samples.

Tensorflow provides the dataset class for loading and processing the floor plan images into tensor objects. It also provides the possibility to define a batch size, shuffle the data within the dataset and map functions to all samples. A perfect shuffling can be achieved, when the BUFFER_SIZE is set to a value greater than the dataset size. Otherwise, the buffer is smaller and the values in the buffer are always filled up by taking the next value in the dataset queue. From the buffer, values are taken randomly. After every epoch during training the shuffle order in the buffer is changed randomly. In order to create reproducible results a random global seed of 42 is set. Algorithm 6.6 creates the tensorflow dataset.

Algorithm 6.6: Loading images into tensorflow datset and setting attributes

```
input_list = glob.glob(str(PATH / '*/*_InputWithAnn.png'))
output_list = glob.glob(str(PATH / '*/*_Output.png'))
dataset = tf.data.Dataset.from_tensor_slices((input_list, output_list))
dataset = dataset.map(load_image_pipeline, num_parallel_calls=tf.data.
    AUTOTUNE)
#Split and augment training data here
train_dataset = train_dataset.shuffle(BUFFER_SIZE,
    reshuffle_each_iteration=True).batch(BATCH_SIZE)
```

Batch Size The batch size is the amount of samples, that are used for one update of all model parameters. The batch size is linked to the learning rate. A small batch size can lead to unstable gradients. To also decrease the learning rate in that case can help. The smaller the batch, the more can the network look into detail of an image, but also the network is more frail to overfitting. The bigger the batch size, the better it can generalize and is more resistant to overfitting. For a better computation efficiency batch sizes to the power of two are tested. When the batch size gets too big, also the RAM capacities of Colab gets to its limit during training. Thus, only batch sizes of 1, 2, 4, 8 and 16 are tested.

Learning rate The 'Adam' optimiser provided by Keras is used for calculating the gradient descent and the according update for each model parameter in a training step. An initial learning rate and two momentum parameters can be set and adjusted.

Amount of epochs/steps One step is one update of the network parameters or in other words one backpropagation step. The batch size gives the amount of samples, after which the parameters are updated. An epoch defines the point at which all samples from the training dataset have been shown to the network once. The amount of steps until one epoch is finished, is calculated by dividing the number of training samples by the batch size. The amount of epochs is calculated as follows from a given amount of steps.

$$Epochs = (Steps * BATCH\_SIZE)/Train\_Samples \tag{6.1}$$

The model is trained using various amount of epochs.

6.3 cGAN-Architecture

The structure of the cGAN model follows the pix2pix architecture proposed by Isola et al., 2016. The basic prediction concept of image to image translation with the pix2pix cGAN is explained in section 2.4.

6.3.1 Generator

The Generator consists of a U-Net-based architecture discussed in more detail in paragraph 2.3.3. The input is a floor plan image with annotations and the output is a predicted image in the same size. In order to extract and translate certain features, the encoder downsamples the image and afterwards a decoder upsamples the pixel into the translated image. With increasing the image size by power of two both in the encoder and the decoder a layer is added. This encoder decoder combination is called autoencoder. The full architecture of the generator with each layer dimension is illustrated in figure A.3.

Encoder The encoder downsamples the pixels and colour channels of the image. One encoder block is created using Keras functionalities and consists sequentially of a convolution layer, batch normalization and the leaky ReLU activation function. The 2D Convolution - for details see section 2.3.1 - uses padding, a stride size of 2 and no initial bias value, that would have been added always on top of the values passing through this layer.

Also, the internal model parameters are initialised using a random normal distribution. The kernel shape of the 2D convolution window is quadratic with a width of 4. Besides the first encoder block, every encoder block is normalized across the mini-batch. And the Leaky ReLU activation function is applied to every block. Algorithm 6.7 shows the creation of one encoder block with Keras.

Algorithm 6.7: Implementation of one encoder block consisting of 2D Convolution, Batch normalization and the LeakyReLU activation function

```
result = tf.keras.Sequential()
result.add(tf.keras.layers.Conv2D(filters, size=4, strides=2, padding='
    same', kernel_initializer=tf.random_normal_initializer(0., 0.02),
    use_bias=False))
result.add(tf.keras.layers.BatchNormalization())
result.add(tf.keras.layers.LeakyReLU())
```

Depending on the original image size the encoder consists of 10 stacked blocks for an input image size of 1024x1024, 9 blocks for 512x512 and 8 blocks for 256x256.

Decoder The decoder upsamples the pixels into the output size and thereby keeping the extracted information. Transposed convolution also called deconvolution is the reversed process of convolution and recreates the original image size from the data string maintaining a connectivity pattern.

One decoder block implemented with Keras functionalities, consists of the transposed 2D convolution, batch normalization, dropout, which is applied only to the first three blocks and the ReLU activation function. Deconvolution also uses a 2D quadratic kernel with the width of 4, a stride size of 2 and padding. The values are also initialized with a random normal distribution. The random dropout rate is set to 0.5. All values, that are not set to zero, are scaled up by $1/(1-rate)$ in order to not change the sum over all inputs. Algorithm 6.8 shows the implementation of one decoder block.

Algorithm 6.8: Implementation of one decoder block consisting of 2D Deconvolution, Batch normalization, dropout and the ReLU activation function

```
result = tf.keras.Sequential()
result.add(tf.keras.layers.Conv2DTranspose(filters, size=4, strides=2,
    padding=’same’, kernel_initializer=tf.random_normal_initializer(0.,
     0.02), use_bias=False))
result.add(tf.keras.layers.BatchNormalization())
result.add(tf.keras.layers.Dropout(0.5))
result.add(tf.keras.layers.ReLU())
```

The blocks are stacked on top of each other with decreasing amount of filters. For an input image size of 1024x1024, 9 stacked blocks are used; for 512x512 there are 8 blocks in usage and for 256x256 respectively 7 blocks. The last decoder block is the same besides shifting the activation function to the tanh function (see figure 2.9d) in order to guarantee, that all output pixel values are in the range of $[-1, 1]$, as the pixel values of all images were normalized to that range. ReLU can also output higher ranges.

Skip Connections With skip connections, layer in the encoder are connected with the respective layer in the decoder, when they have the same dimensionality. The values are concatenated as explained in subsection 2.3.1. In figure A.3 the skip connections and the concatenation dimensions are illustrated. The random skip connections should enable a more direct way of passing information from layer located close to the input to layer close to the output.

Optimiser The gradient for model parameter updates during backpropagation is calculated using Keras built in ’Adam’ optimiser. The stochastic gradient descent method is explained in detail in subsection 2.2.3. It adapts the learning rate during training and also applies a parameter specific learning rate based on the first moment, which is the mean of the gradients and based on the second moment being the uncentered variance. The decay parameters β_1 and β_2 manage the rate of decay of the past averages. In Table 6.2 all parameters are stated with their initial values, that are used in the model.

Table 6.2: Summary of parameters used in Adam optimiser

Parameter used in Keras Adam Optimiser			
Parameter Name	Symbol	Value	Purpose of Parameter
Learning rate	ν	$2e-4$	Original learning rate
Beta 1	β_1	0.5	Decay parameter for first momentum (mean)
Beta 2	β_2	0.999	Decay parameter of second momentum (variance)
Epsilon	ϵ	$2e-7$	Constant for numerical stability in denominator

Loss function The loss or also called objective function is the summation of the binary cross entropy loss, explained in paragraph 2.4.3 and the L1 distance of the prediction and the target image, which is summarised in paragraph 2.4.3. The binary cross entropy loss is calculated using the classification into real or fake parts of the predicted image made by the discriminator and an array of ones. Figure 6.3 illustrates the calculation pipeline and components of the generator loss.

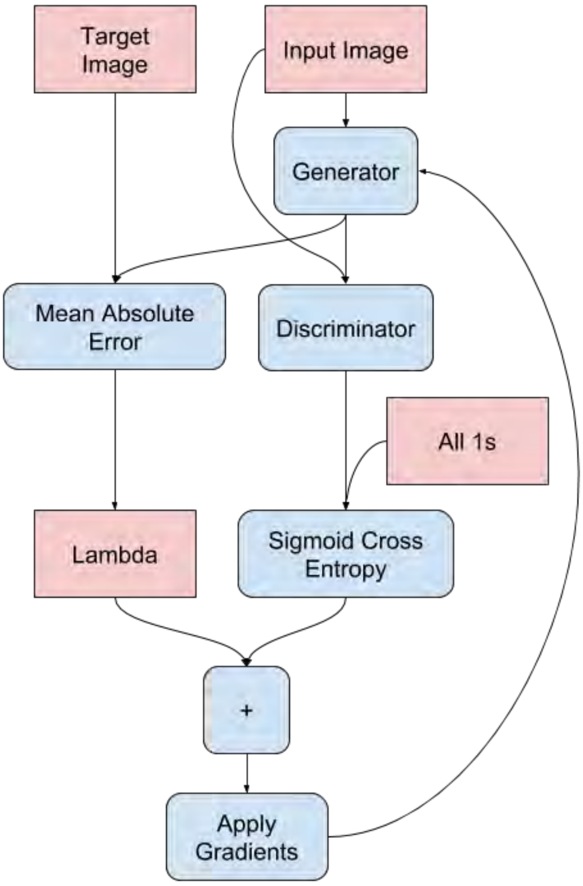


Figure 6.3: Components of the Generator Loss calculation

Lambda is a parameter that is multiplied with the L1 distance in order to give it more or less weight in the objective function gradient calculation. For implementing the generator loss, Keras functionalities are applied for calculating the sigmoid cross entropy loss. Algorithm 6.9 shows its implementation. The Adam optimiser calculates the gradient using the loss as objective function and updates the parameters of the model.

Algorithm 6.9: Calculation of the generator loss consisting of the binary cross entropy loss and the L1 distance

```
# Sigmoid cross entropy loss calculation
loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gan_loss = loss_object(tf.ones_like(disc_generated_output),
    disc_generated_output)
# Mean absolute error being the L1 distance
l1_loss = tf.reduce_mean(tf.abs(target - gen_output))
total_gen_loss = gan_loss + (LAMBDA * l1_loss)
```

6.3.2 Discriminator

The discriminator consists of a convolutional PatchGAN classifier architecture explained in more detail in paragraph 2.3.3. The input of the discriminator are two concatenated images; in case of this model the input is either the concatenation of the input floor plan with the target image or the concatenation of the input floor plan with the prediction image of the generator. As output it classifies overlapping 70x70 patches of the concatenated images as real or fake.

Architecture The first layer concatenates the two input images. The next blocks downsample the concatenated input images. Each block consists of a 2D convolution layer, a batch normalization, which is turned off only in the first block and the leaky ReLU activation function. Depending on the size of the input images, either 3 (for 256x256), 4 (for 512x512) or 5 (for 1024x1024) downsampling blocks are used.

Algorithm 6.7 shows the implementation of the downsampling block, which is applied both in the generator and the discriminator. Again, a convolution window of 4x4 is used. As next step a layer is applied to the data, where 2D zero pads with a thickness of 1 are added at the top, bottom, left and right of the downsampled images. Then comes again a block of 2D convolution with a stride of 1, batch normalization and the leaky ReLU activation function. Finally another pad of zeros is added and the 2D convolution with a stride of 1 is applied. The output is a binary patch with the size of 30x30. Each entry in the binary patch classifies an overlapping patch of 70x70 pixels of the concatenated input images as real or fake. The full layer structure of the discriminator is illustrated in the appendix in figure A.2.

Optimiser The Discriminator also uses an Adam optimiser in the same way, as the generator does. Also, the parameters stated in table 6.2 are used for implementing the Adam optimiser for the discriminator.

Loss function For calculating the loss of the discriminator, also the same sigmoid cross entropy loss function is used as in the generator. The loss is calculated for a real image, being the target image of the network and for the generated image, that has been predicted by the generator.

’real_loss’ is the sigmoid cross entropy loss between the real image and an array of ones, as the discriminator should classify every patch of the target image as real, which is a one in binary classes. ’generated_loss’ is the sigmoid cross entropy loss between the generated image and an array of zeros, as the goal of the discriminator should be to categorize every patch as fake.

The total discriminator loss is the sum of both losses. The loss calculation is implemented according to algorithm 6.10 and the training procedure is visualised in figure 6.4.

Algorithm 6.10: Calculation of the Discriminator loss consisting of the binary cross entropy loss of a real and a generated image

```
loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)
real_loss = loss_object(tf.ones_like(disc_real_output),
    disc_real_output)
generated_loss = loss_object(tf.zeros_like(disc_generated_output),
    disc_generated_output)
total_disc_loss = real_loss + generated_loss
```

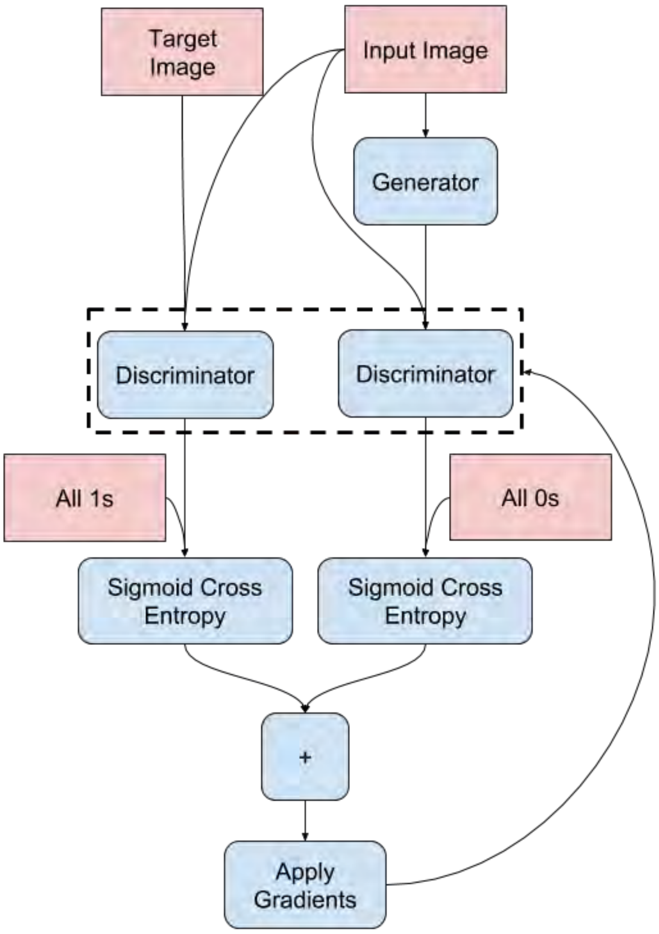


Figure 6.4: Components of the Discriminator Loss calculation and training workflow

6.4 Training

After preprocessing the dataset, creating the architecture of the generator and the discriminator and defining the loss and optimiser, the training procedure is implemented.

6.4.1 Training workflow

The implementation of training is basically a for-loop over the training steps. In every step the loss for the discriminator and the generator as well as the gradient are calculated. The parameters of the cGAN model are updated after every batch of floor plan images, which means, that the generator and the discriminator are both optimised at every step.

Every ten steps the losses are stored in order to create a training loss curve with the help of tensorflows built-in visualization tool tensorboard. The loss is calculated both on the training as well as on the validation data set in order to compare the training process and prevent overfitting. Checkpoints for restoring the model parameters of a certain point in training are stored every 50,000 steps. A sample image from the test dataset is used for observing the visual changes in the model prediction. Every 1000 steps the test images are predicted by the generator and stored in order to follow the internal learning steps of the model.

6.4.2 Selection of parameter combinations in each training run

The hyperparameters are tuned by educated guessing and trial and error, as an automated hyperparameter optimiser would have an unfeasible computation time. In every new training attempt only one parameter is changed in order to understand its effect on the results. The parameter combinations for the training runs are summarised in table A.1. When selecting the combinations the following parameters were taken into account.

1. **Batch size:** to not exceed the RAM, batch sizes of 1, 2, 4, 8 and 16 are tested
2. **Image size:** 256x256, 512x512, (1024x1024)
3. **Data Augmentation:** Left-right flip, Upside down flip, all rotations and combinations
4. **Training steps:** 20,000 to 350,000
5. **Learning rate:** see table 6.2; default parameters
6. **Lambda:** 10; 100; 1000; 10,000
7. Train only generator without discriminator loss (use only L1-distance)
8. **Shuffling of dataset:** Train with and without random shuffling

6.4.3 Encountered Problems

As the model is trained using google colab, the computation time and RAM is limited. Even though an update to google colab pro increases the allowed runtime and the available RAM, training with a floor plan size of 1024x1024 is not possible as it exceeds the RAM capacity. Also, training the model with a batch size bigger than 8 leads to a collapse due to RAM shortage.

Furthermore, the model can only train, while the browser window is open and in regular intervals a verification needs to be confirmed. Otherwise, the model stops training and looses the already learned progress when the current runtime disconnects. The runtime also crashes, when the computer is shut down or goes into stand-by mode.

Another problem is the allocation of GPU by colab. When using lots of computation time in close intervals, colab allocates the model to no GPU anymore. The computation time is then so high, that the model cannot finish training within the possible runtime of 24h provided by colab.

When using a colab machine for training, the local storage cannot be accessed. Thus, the checkpoint data, intermediate prediction images and the loss data are written to files in the google drive storage, which provides a capacity of 15 GB. Especially the checkpoint data files have a big filesize. When the drive storage gets full during a run, the code does not write any more data, leading to missing results in the ongoing training process.

Chapter 7

Evaluation

As explained in subsection 2.4.5 there exist several evaluation metrics for GANs. It is still most common to evaluate the model performance by visual inspection. Coinciding with the aim of this thesis to produce a good prediction of the translated floor plan, it makes sense to use visual inspection in the evaluation of the prediction results of the proposed GAN. The performance of the different parameter combinations is evaluated visually and by looking into the loss curves. Further, the loss curves of the generator and the discriminator are analysed and some phenomena in the loss curves are discussed.

7.1 Evaluation based on loss curves

Every ten training steps a new value for the loss is stored. The loss is calculated for both one training and one validation sample per step. The samples within the training and the validation set shuffle randomly in every calculation step leading to a variation in the order of the samples. After every sample in the dataset was chosen for calculation, random shuffling starts again, so that every sample is fed once to the network in every epoch in changing order. Shuffling is done using a global seed in order to get reproducible results.

7.1.1 Loss curve shapes

The loss of the generator is split into the L1-distance and the generator GAN loss, which is the penalty given by the discriminator. The discriminator has its own loss function. In the following, the three curves are discussed and explained. Every Loss curve is smoothed using a rate of 0.95 or 0.99. The loss curves are shown for a run with a size of 512x512, batch=1, the default ADAM optimiser, Lambda=1000 and all data augmentation steps.

Generator L1-Loss The L1-Loss is an important curve for the evaluation of this model. It shows the distance between the prediction and the target floor plan image. The aim is, to get a prediction as close as possible to the target and thus, to decrease the L1-Loss as much as possible.

Theoretically, the L1-Loss curve should in average decay exponentially. In the beginning, the model learns very fast to translate the pixel and therefore the training progress is quick. After the first few steps the learning is rather slow as the model is learning small details. Further oscillations in the learning process show, that the model tries various translation methods in order to find the global minimum in the loss function. As soon, as the average loss increases for some epochs, training should stop.

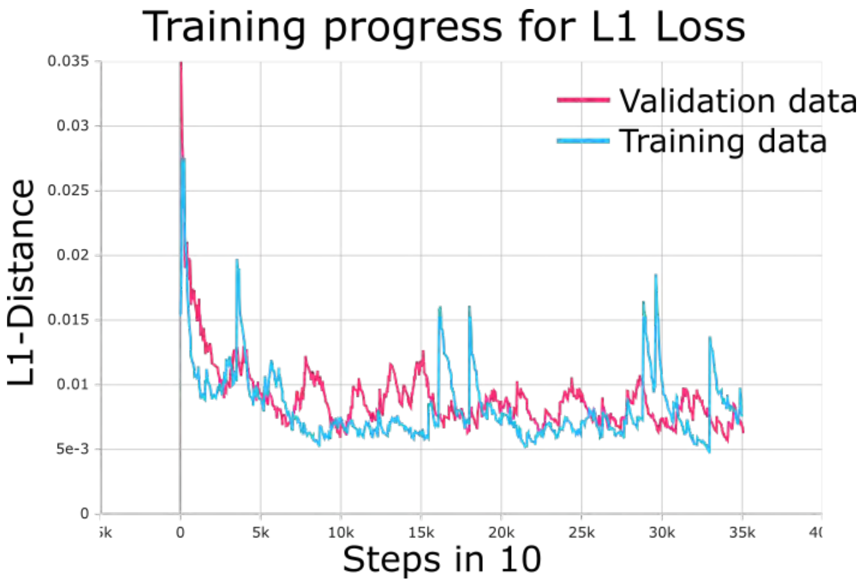


Figure 7.1: L1-Loss curve of generator for training and validation set

Figure 7.1 shows a typical loss curve for a good training procedure. The training loss is lower than the validation loss curve and the gap between both curves is in average not increasing.

Generator GAN Loss In a good training progress, the penalty given by the discriminator should oscillate around a value without going up or down too much. If either the generator GAN loss or the discriminator loss get low, one model is dominating over the other. This means, that the whole GAN model is not trained well and the training progress is slowed down. According to Isola et al. a value of $\log(2) = 0.69$ is a good reference point for both losses ISOLA et al., 2016. This loss value implies, that the discriminator is in the average equally uncertain about the two options of classifying the floor plans as real or fake.

Discriminator Loss The discriminator loss rates the performance of classifying an image as real or fake. As explained in the previous paragraph the optimal loss oscillates around a value without a trend to go up or down. Increasing the weight on the L1 loss also increases the gap between the training and validation loss curve of the discriminator.

7.1.2 Phenomena in Loss curve

Oscillations The loss curves for both the generator and the discriminator show oscillations. The objective function should be minimised. High oscillations can be a sign of overshooting local minima, due to a too high learning rate.

Oscillations can also indicate, that the model is training on a very inhomogeneous dataset. In every floor plan different amount of pixel need a translation, leading to variations in the loss curve. Averaging for example the loss of 10 floor plans for each loss point in the curve would help avoiding strong oscillations and show the overall trend better.

Smoothing of Curve All loss curves have many outliers and show strong differences in each sample. A reason for the outliers is the high in-homogeneity and also noise of the dataset samples. Some samples have more handwritten symbols, that lead to more pixel, that need a prediction; others have only one or two symbols. Accordingly, the difference between the target and the prediction is bigger, when more pixel need a translation and cannot just be copied.
Also, the size and the amount of geometric features within the floor plans differ a lot. Sometimes handwritten symbols might overlap or are so close to each other, that the model is not able to isolate the symbol.

In order to understand and visually compare the overall trend of the loss curves, the curves are smoothed using a tensorboard internal functionality. To get the best comparability, a smoothing value of 0.999 is chosen for further plots. Smoothing means, that all points within the curve are averaged. When a smaller value for smoothing is used, the oscillations in the loss become visible.

Gap between Train and Val curve When plotting both the loss curve of the training and the validation data, the validation loss is usually higher than the training loss, as the network already knows the training data and is therefore adapted to this data, but is not trained using the validation data. An increasing gap between both curves can be a sign of overfitting, especially, when the loss of the validation data is rising again, but the training loss is still going down.

7.2 Results

A test set for analysing the prediction capacities and limitations of the GAN is additionally created. The 4 test floor plans include all symbol classes, that are trained in the model. The floor plan images were created following the creation pipeline in section 5.2. As many symbols are combined within one test sample, the test floor plans pose a challenge for the model. The test dataset is designed to bring out the networks limitations.

Attached are all four test images in the size of 512x512. The first test sample in figure A.4 combines various operations. The second sample (figure A.5) includes all modification symbol classes, the third sample (figure A.6) combines all symbols belonging to the add operation and in the fourth image (figure A.7) are all trained delete operations. In the following, the influence of different parameters are analysed visually using the test images and also the general capabilities and limitations of the model are discussed.

7.2.1 Influence of Parameter tuning

In order to optimise the performance and accordingly the prediction accuracy of the model, hyperparameter tuning and data augmentation are inspected. When changing hyperparameters and adding data augmentation, some visible improvements could be achieved. For visual inspections of a certain hyperparameter always the same parameter combinations are used in one comparison part and only one hyperparameter is changed.

Data augmentation Data augmentation results in an obvious improvement in the ability of the model to generalize and project symbols. The L1-loss is over all clearly lower for the validation dataset in a training process with all data augmentation steps than in a run using only the original dataset without any augmentation. Figure 7.2 shows the according validation L1 loss curves of both training processes. The oscillations in this graph are rather low and accordingly the absolute L1 loss value for the val-set, as in this comparison a local seed was set instead of a global seed.

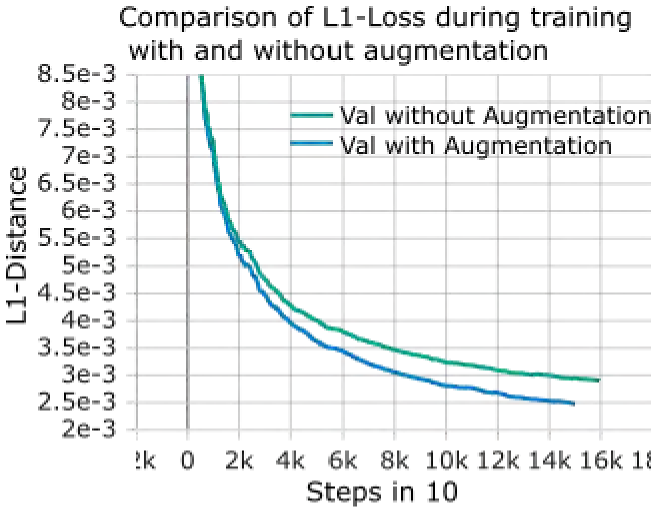


Figure 7.2: L1 loss curve of generator for the validation set of a run with full data augmentation and a run without any augmentation steps

Figure 7.3 highlights some significant improvements in the prediction accuracy like the label deletion or removing and adding the wall (for target image see A.4).

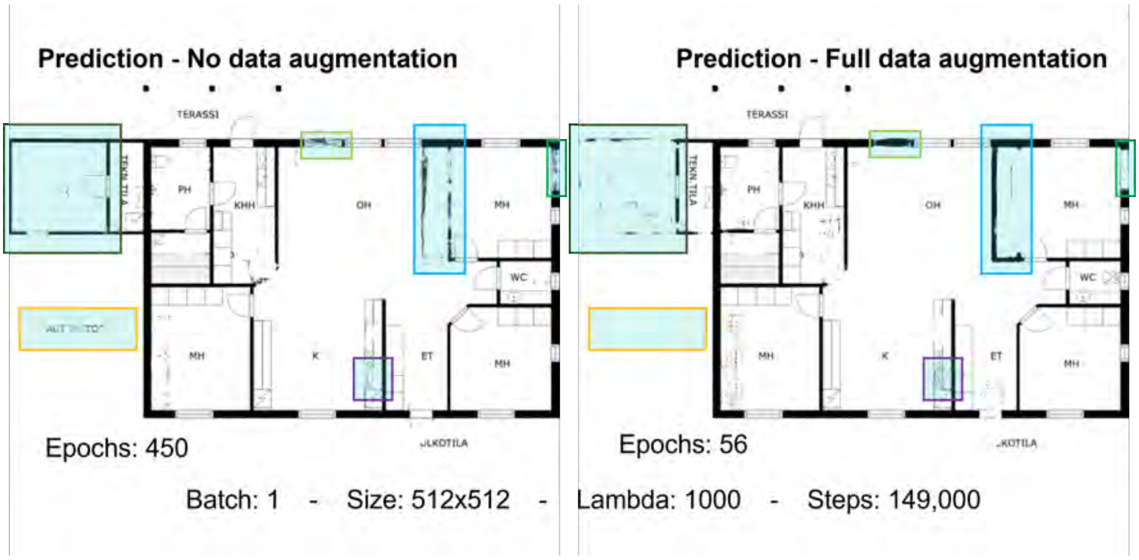

Prediction - No data augmentation
Prediction - Full data augmentation
TERASSI
TEKN. TILA
PH
KHH
OH
MH
WC
MH
K
ET
MH
ULKOTILA
Epochs: 450
Epochs: 56
Batch: 1 - Size: 512x512 - Lambda: 1000 - Steps: 149,000

Figure 7.3: Prediction of first sample in test set of a model trained on a dataset without and with all data augmentation steps

Image size The selection of the floor plan resolution makes a slight difference in the generator performance. The model is more accurate in predicting small symbols and thin lined furniture elements, when the resolution of the images is better. With better resolution the lines are sharper an not that blurred, visible in the attached test sample comparison in A.8. The validation L1 loss curve of both training runs shows a similar shape (cf. A.9). Here, also in both runs a local seed is set, leading to a low absolute loss value.

Training steps Figure A.10 shows the visual training progress in steps of 50,000 for the test sample 1. With more epochs the model can learn smaller details and can predict sharper digital symbols in a more detailed representation. But can also forget prediction abilities, that already worked well. This is for example visible in the prediction of the kitchen line or the full room deletion.

The model trains in an oscillating way, which becomes also clear when looking into the L1 loss (cf. 7.1). In some steps the prediction ability of the model is better and in other steps the prediction becomes worse again. Accordingly, also the translated floor plans appear in a better and in a worse translation accuracy.

Another problem is, that some symbols are accurately predicted in a certain step, when other features are translated bad and vice versa in another step. Thus, it is hard to tell the best amount of epochs for the full prediction. Instead, the best amount of epoch should be stated for each individual symbol translation task. Additionally, the optimal epochs are strongly connected with the other hyper parameters and accordingly change when another parameter combination is used.

Training should stop at latest, as soon as the validation loss curve of the generator starts to rise in average for some epochs.

Weight of L1-Loss The objective function of the generator sums up the L1-distance between the prediction and the target image as well as the output quality classification of the discriminator. The parameter λ sets the weight in the generator loss calculation of the L1-distance. The loss function of the generator should accordingly profit from both the GAN advantages of learning how to predict real images and the CNN advantage to minimize the distance between the target and the prediction images. The higher λ the more influence has the L1-distance in the loss calculation.

One important measure of the prediction quality is to minimize the L1 loss and thus to minimize the difference between the prediction and the target floor plan. The more weight is given to the L1 distance, the better is its optimisation. Training the generator only on the L1 loss without any discriminator classification results in a CNN model.

The visual comparison in figure 7.4 shows clearly a better performance in the prediction accuracy of the CNN model than of a GAN with $\lambda = 100$ when training for 150,000 steps. Some obvious differences are marked in 7.4 like for instance the prediction of blue pixel by the GAN, several delete operations, which are not executed by the GAN and the fine prediction of deleting and adding windows and walls by the CNN.

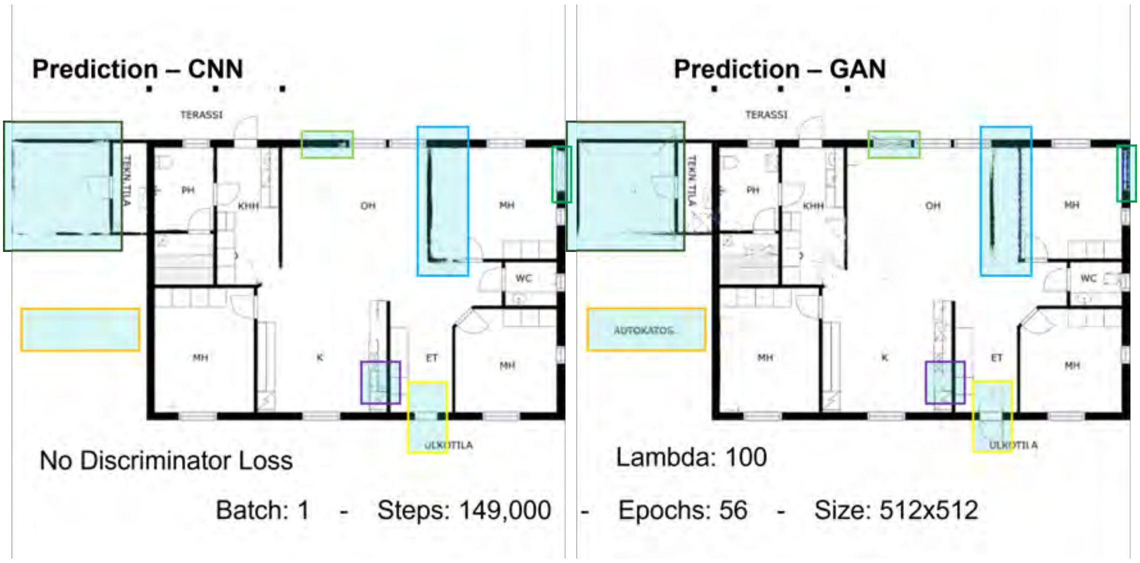


Figure 7.4: Visual inspection of pure CNN prediction in comparison with GAN prediction with a L1 loss weight of 100

The L1 loss curves (cf. 7.5) of models with different λ also state clearly, that the average L1 loss decreases with higher weight. The analysis of the L1 loss curve and the visual inspection both support the supposition of an improving performance with higher weight for this dataset.

Looking into the discriminator loss curve (cf. A.12) and the generator GAN loss curve (cf. A.11) for low lambda and for no discriminator loss, the discriminator clearly outperforms the generator loss. The best performance of the Gen GAN loss is for lambda=100, where the visual inspection clearly shows a bad prediction performance.

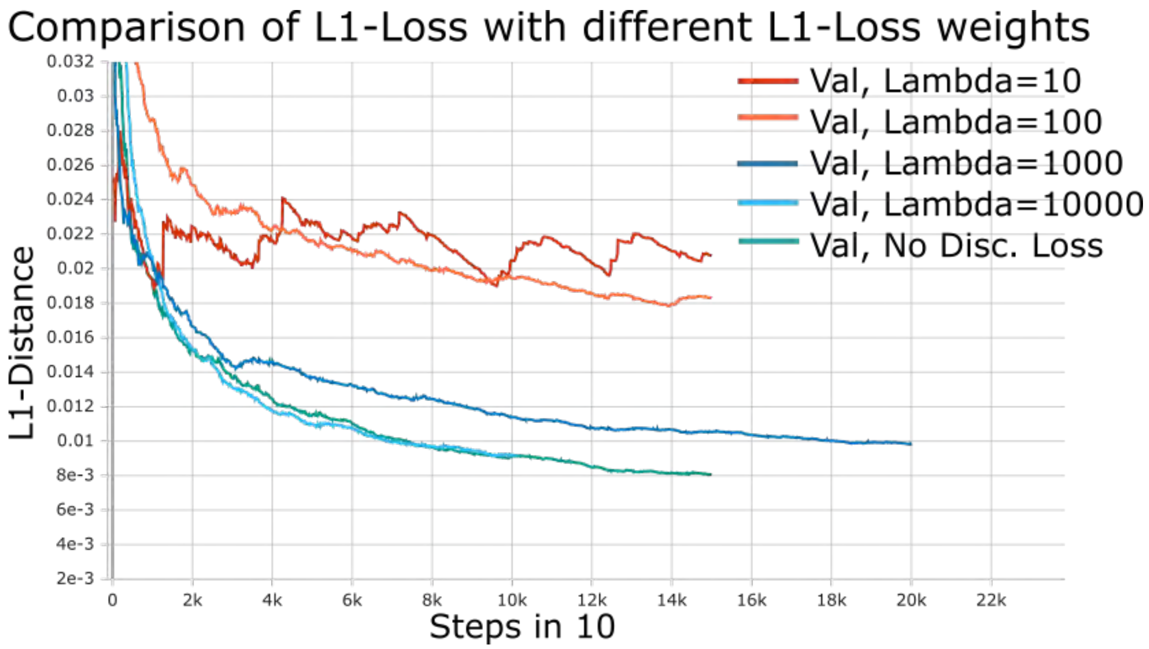


Figure 7.5: Difference in validation L1-Loss curve for different weights λ of L1-Distance in generator loss function

Comparison with CNN Event though the visual prediction of the CNN is, according to the previous paragraph, with this parameter combination and for this dataset clearly better, still the advantages of the GAN can be of use in this application. When training longer with a bigger and more diverse dataset, the advantage of the GAN to predict real features and to generalize can be essential.

Then, the CNN might only be able to predict a digital symbol, which seems to have a similar pixel combination close to the target symbol. But in fact it looks completely different and fake from the point of view of a human being. At this point the penalty given by discriminator might lead to a prediction change in order to translate this symbol more 'real'. Accordingly, the discriminator in combination with the generator might be able to push the generator loss from a local minimum, where it got stuck as a pure CNN, into a global minimum.

When training the network with the aim to balance the discriminator and the generator loss, maybe the advantage of GANs to generalise can also be used more in this application. Therefore another hyperparameter combination and training sequence between the generator and discriminator might be needed.

Optimiser - Learning rate A higher learning rate theoretically improves the loss after less training steps, but brings the danger of overshooting a minimum. A high learning rate in combination with a slow decay of the first and second momentum can prevent overshooting, as it decelerates learning, when fine details are learned. Comparing the validation L1 loss for runs with different learning rates, a higher learning rate shows clearly a quicker improvement of the loss (cf. 7.6). Also, a slower decay of the first momentum shows a better learning.

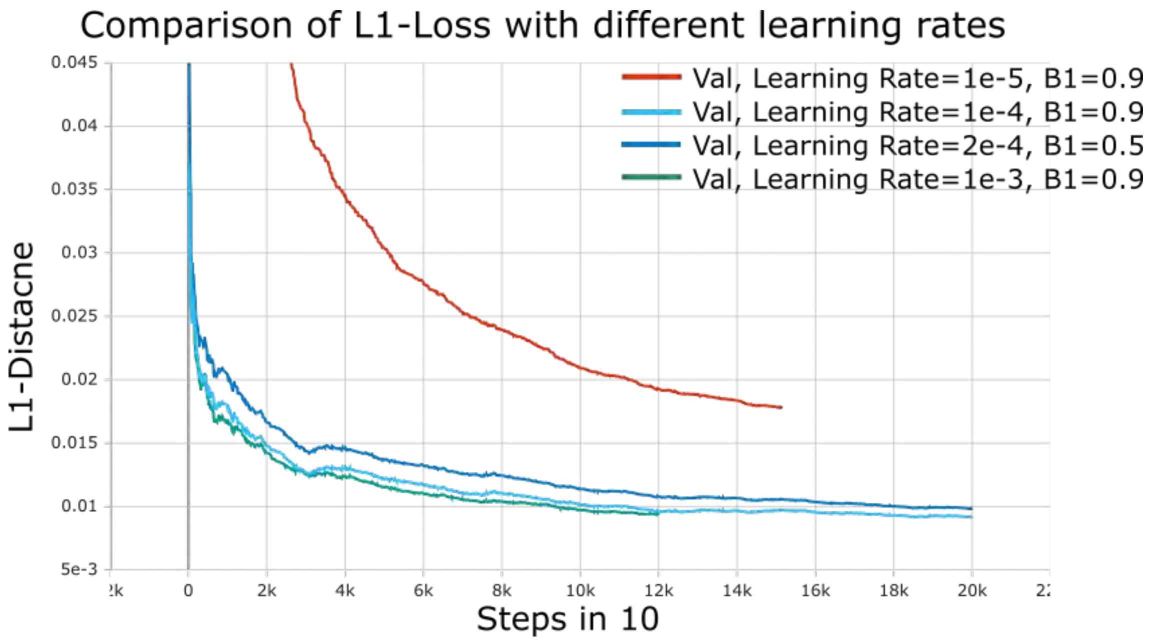


Figure 7.6: Difference in validation L1-Loss curve for different learning rates

Batch size When the batch size is increased, an epoch consists of less steps. The model parameters are updated less often within one epoch and one gradient descent step is theoretically directed more into the direction of the minimum of the objective function.

Also, the computation time for one step rises, leading to more or less the same amount of time for training the model on a full epoch compared to a lower batch size. The loss and visual output do not change significantly when changing the batch size. The time dependent L1 loss curves for the validation set with changing batch sizes show a slight improvement in learning for smaller batch sizes (cf. 7.7) using the adam optimiser with the parameters stated in table 6.2. The loss curve shows, that the average gap between the validation and training set also gets slightly bigger with increasing batch size.

The validation set shows more outliers than the training dataset, when the batch size is bigger than 1. The reason for this phenomenon is, that the loss is calculated on the average of all floor plan samples within the batch leading to averaging the outliers. The validation loss is always calculated using a batch size of 1.

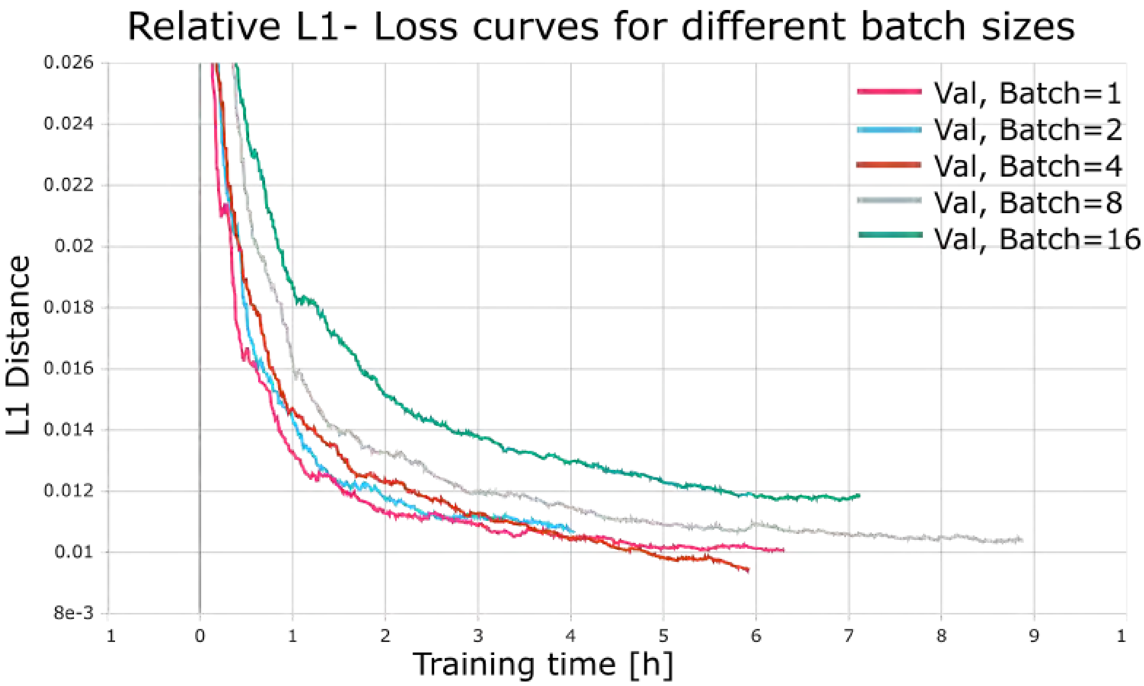


Figure 7.7: Difference in time dependent validation L1-Loss curve for different batch sizes

Shuffling Changing the order of the dataset samples within each epoch both in the training as well as in the validation set leads to bigger oscillations in the loss curves and therefore also to a higher loss. Shuffling creates more convincing and more comparable loss curves. The average validation loss is higher than the training loss. This should always be the norm, as the network already knows the training data. Validation data is unknown to the trained model and therefore the overall prediction performance on validation data should be worse. This is visible in figure A.16. Visually there is no obvious improvement between a prediction with and without shuffling (cf. A.15).

7.2.2 General Capabilities and Limitations of the network

The model is able to copy the original parts of the floor plan, where no modification symbols are applied, very well after a short amount of training steps. Especially walls, windows and doors are copied with a high accuracy. The model struggles especially in the first training steps with translating original furniture elements. This is visible in the training progress in figure A.10, when looking into the prediction after step 1001.

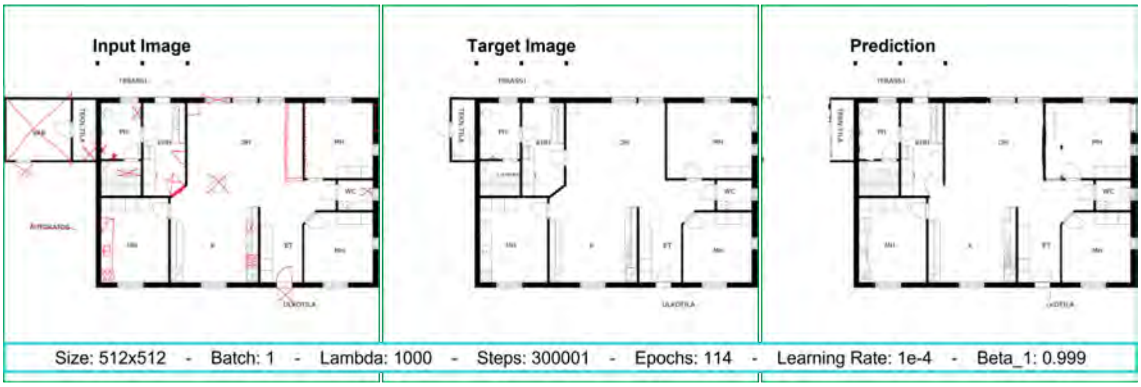
When translating handwritten symbols, the network copes better with symbols, that are represented more often within the training dataset and struggles with complex symbols like adding a sauna bench or a chimney. The neighbourhood plays a great role in the translation quality of pixels. Isolated symbols are predicted with a higher accuracy than overlapping symbols or symbols being located spatially close to other feature elements.

Also, the placement within the floor plans is important. Adding elements or deleting elements close to walls can confuse the network in a way, that it deletes wall pixels as well.

Pixels in the area of a red handwritten symbol are blurred in the beginning and get overall sharper with more training steps as the prediction accuracy gets better.

The network realises, that most walls are either horizontal or vertical. When there are any changes within a non horizontal or vertical wall, the network struggles with tasks it is normally able to predict with a better accuracy. This becomes clear when looking into the prediction of deleting a door in the test image 1 in a non horizontal or vertical wall (cf. 7.8).

Finding an epoch where the model has learned to translate all symbols as best as feasible in every floor plan sample is not possible. Accordingly, for all test and validation image predictions the output after 300,000 steps is chosen with the same parameter combination to provide the possibility to compare the results better (cf. test sample 1: 7.8; test sample 2: 7.11; test sample 3: 7.10; test sample 4: 7.9; also compare with prediction of validation set in the Appendix A).



Size: 512x512 - Batch: 1 - Lambda: 1000 - Steps: 300001 - Epochs: 114 - Learning Rate: 1e-4 - Beta_1: 0.999

Figure 7.8: Prediction of test sample 1

Delete The symbol for deleting elements is always a cross. Even though deleting different elements, the symbols as well as the action to delete the black pixels is the same operation for the model. This leads to more symbol samples in the full dataset compared to add or modify symbols. Accordingly, the network learns simple delete operations quite fast, reliable and with a rather good prediction accuracy.

If the feature symbol is isolated and not too big or too small, the delete operation is predicted well. Also, deleting windows and instead adding a wall is translated with a high accuracy. Full room deletions or removing furniture elements with a high width to height ratio like for example shower walls are predicted with a low accuracy. The model also struggles with deleting elements close to walls, to other elements or close to other handwritten symbols.

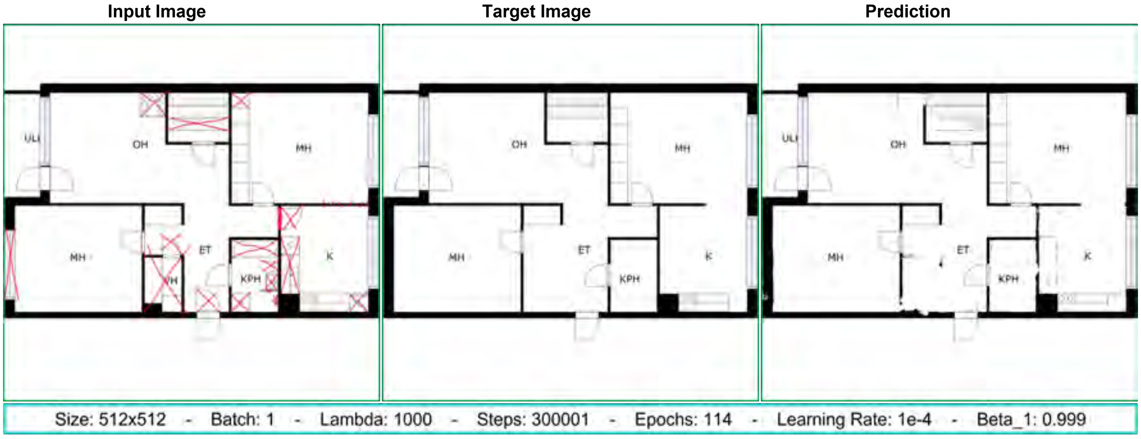

Input Image
Target Image
Prediction
ULI
OH
MH
MH
ET
KPH
K
Size: 512x512 - Batch: 1 - Lambda: 1000 - Steps: 300001 - Epochs: 114 - Learning Rate: 1e-4 - Beta_1: 0.999

Figure 7.9: Prediction of test sample 4 including all delete symbols

Add The translation quality of an add-symbol correlates strongly with its occurrence frequency within the dataset and the complexity of the symbol. For instance, there exist 192 cases of adding a window and 168 cases of adding walls within the full dataset. After several training steps the model is able to predict both with a high accuracy. With other symbols like adding a full kitchen line or a bathtube with 33 or respectively 59 representations in the dataset, the network is struggling and predicts blurred shapes.

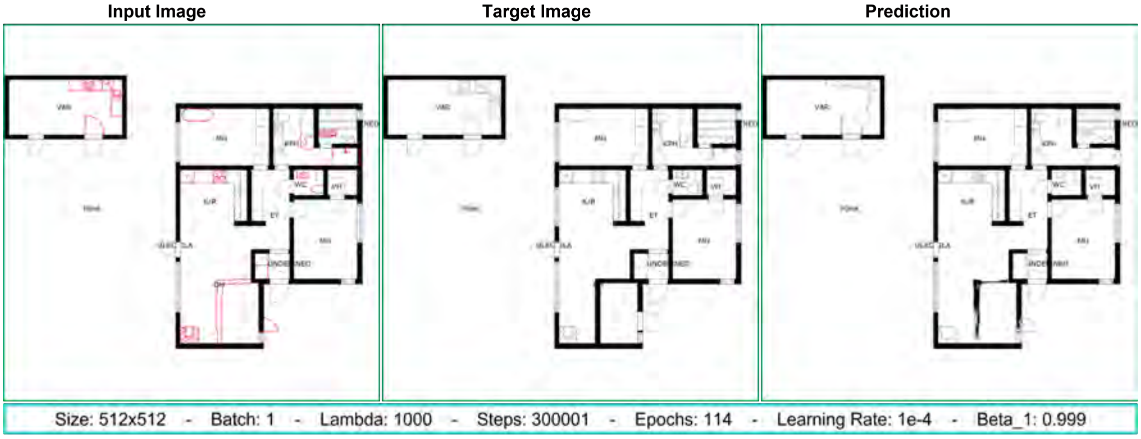

Input Image
Target Image
Prediction
Size: 512x512 - Batch: 1 - Lambda: 1000 - Steps: 300001 - Epochs: 114 - Learning Rate: 1e-4 - Beta_1: 0.999

Figure 7.10: Prediction of test sample 3 including all add symbols

Modify All modification operations are represented worst in the dataset. Accordingly, no parameter combination is able to predict a modification correctly. Much more symbols are necessary within the dataset in order to get a result. Either the network just deletes the red handwritten pixel and copies a blurred area or it copies the symbol into greyscale pixel without modifying the actual element. Changing the characteristics, scale, orientation or placement of any feature element provides the most complex modification annotation and therefore necessitates a much higher representation in the dataset.

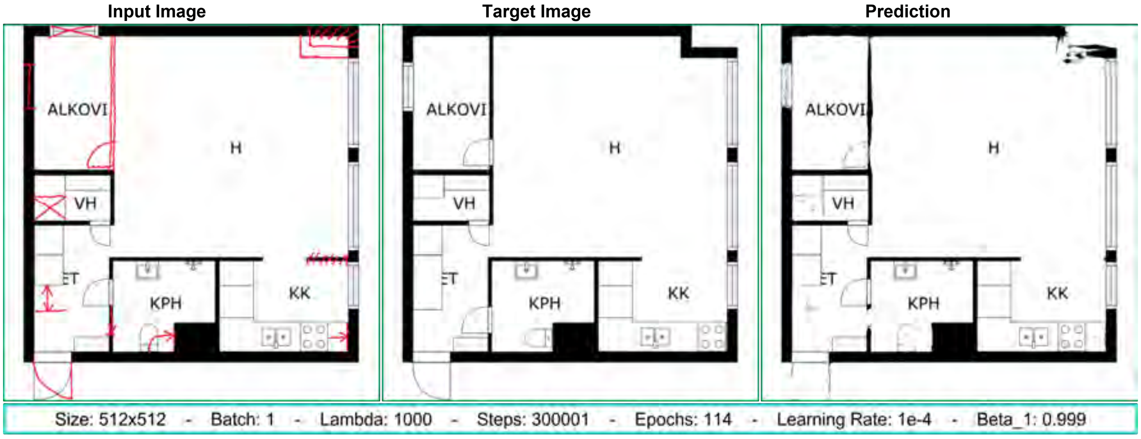

Input Image
Target Image
Prediction
ALKOVI
H
VH
ET
KPH
KK
ALKOVI
H
VH
ET
KPH
KK
ALKOVI
H
VH
ET
KPH
KK
Size: 512x512 - Batch: 1 - Lambda: 1000 - Steps: 300001 - Epochs: 114 - Learning Rate: 1e-4 - Beta_1: 0.999

Figure 7.11: Prediction of test sample 2 including all modification symbols

7.3 Improvements

Regarding the dataset some additional improvements are possible in order to raise the performance of the model prediction. Also some other aspects pertaining to the GAN architecture can further be adapted, tested and evaluated.

7.3.1 Dataset adaptions

The correlation between the improvement in prediction accuracy of a symbol and its occurrence frequency within the dataset leads to the assumption, that a bigger and more diverse dataset is needed for a good translation performance. This section provides some improvement ideas for more diversity and quantity.

Data augmentation In order to create more data samples and help the model to generalise better, more data augmentation steps could be applied to the dataset. A variation in the floor plan size, random cutting without intersecting a handwritten symbol or colour changes are some more possible augmentation steps. Also creating much more individual samples, which are still consistent with the creation pipeline, will surely increase the model performance and its ability to generalize.

Colour of handwritten symbols Another interesting attempt could be to modify the colour of the handwritten symbols in the input floor plans in such a way that all symbols belonging to a certain operation group have the same colour. For instance, all delete symbols keep the red colour, all adding operation symbols become green and all modification symbols are changed into blue pixel. This might also improve the model performance, as learning might become easier through this additional classification.

It also provides more options for training models on the same dataset with different prediction purposes. When removing for instance all green and blue pixel and the according digital symbols in the target image in a preprocessing step, only delete modifications will remain in the dataset. Subsequently models focusing only on one operation type can be trained with the same dataset. A model focusing only on delete operations will probably perform better and remember smaller details than a model, that knows how to translate all symbols at once.

Greyscale floor plans Converting all images into greyscale or even binary images, including the handwritten changes, reduces the computation effort, as there is only one colour channel instead of three, that the network needs to take care of. This might lead to a decrease in the prediction performance, though.

Diversity in handwritten symbols Also creating more diverse data samples is an essential step in improving the model performance in a way, that it can be used in real world applications. New handwritten symbol classes like for instance adding a balcony or scaling feature elements can be added. In order to fit the model to real world handwritten modifications in floor plans, the dataset should be extended in a way, that different peoples handwriting is used for all symbols. Further, different pencils for drawing the symbols in the floor plans should be taken in order to get a diversity in line thickness, colour and transparency of the annotations.

Diversity in floor plan style Different software produces different digital drawing excerpts with varying focuses and LODs. Depending on the construction step these different floor plans are also used on the construction site leading to people adding handwritten annotations. Accordingly, a diversity in the style of the floor plan images also increases the ability of the GAN model to generalise and is therefore an essential step towards a commercial usage. The CubiCasa5K dataset also provides around 300 real world samples with various styles and handwritten modifications, which can also be added to an increased dataset.

7.3.2 Possibilities for GAN model upgrades

Even though hyperparameters were already tested using educated guessing and trial and error, further hyperparameter tuning and some model architecture upgrades might improve the performance of the GAN. Turning ones attention more to optimising the Generator GAN loss and the discriminator loss could bring out more the advantages of GANs to generalize better.

Amount of Layer Adding layers within the generator and discriminator could lead to an improved learning as it might be possible, that the network remembers the translation of smaller details better. Also removing layers could be worth an attempt in order to prevent some overfitting phenomena. It is essential to find a good ratio between adding and deleting layer in order to not have over or underfitting.

Training the network on a finer image resolution like for example 1024x1024 could also result in a better translation of symbols as more detailed features might be captured. Therefore a better machine with a bigger RAM capacity is necessary.

Discriminator adaptions As the discriminator loss becomes smaller and smaller with ongoing training and the generator GAN loss increases steadily, it probably outperforms the generator. In order to slow down the learning progress of the discriminator in comparison to the generator, a possible improvement for the overall prediction could be, that the discriminator updates its parameters not within every step but only for instance after every fifth step.

Another attempt to improve the discriminator performance in ranking the generator is to change the size of the discriminators classification patches. On the one hand decreasing the size leads to finer classification and gives more weight to small details. On the other hand the neighbourhood and therefore captured feature size decreases as well. A good compromise between increasing and decreasing the patch size is essential in order to enable the discriminator to be a good assistance in penalising the generator at the right spot.

Post processing Post processing of the predicted floor plan images will increase the image quality. Resizing the image to the original size, refining the drawn lines, deletion of noise in the output images and straightening added paths are some possible processing suggestions.

Chapter 8

Conclusion

The motivation of this thesis is to automate the time consuming and error prone process of transferring annotations in analog floor plans into the according digital version providing consistency. The results show a proof of concept that cGANs can learn the pixel wise translation of handwritten symbols into digital symbols embedded in floor plans.

The H-Symb FP dataset created within the scope of this thesis provides enough samples for letting the model learn how to delete isolated feature elements within the floor plan. Adding and deleting walls and windows shows a high prediction accuracy in many output images. However, for more complicated operations like adding a furniture element, rotating an element or modifying elements spatially close to other feature elements, additional data samples are necessary for sufficient training. As a result of this study, a correlation between the occurrence frequency of handwritten symbols and the overall quality of their prediction accuracy is detected.

When compared with the cGAN, the results of training the generator as standalone CNN without influence of the discriminator show, that the training process is accelerated and the prediction quality is improved. Accordingly, when trained on the H-Symb FP dataset the advantages in using a GAN model instead of a CNN could not be detected. However, in a commercial usage GANs can still be advantageous when a more extensive and diverse dataset is available for training. On the basis of more samples the GAN can learn to differ better between real and fake symbols and accordingly improve its ability to generalize.

Improvements in the dataset, like more training samples and more diversity in the handwritten annotations as well as in the drawing layout, and the addition of some image post-processing in order to refine the pixel of the predicted symbols can bring the model to a point, that it can be integrated into commercial construction processes.

In some future work a graphical user interface (GUI) can be implemented in order to provide an easy and understandable handling of the floor plan translation process. This GUI can even be embedded into an application on a mobile phone or touch pad, so that the camera can scan the floor plan image with handwritten symbols directly on the construction site and return the translated image. Both office and construction workers benefit from this tool, as it helps to avoid misconceptions in handwritten symbols and is easy to use in every situation.

As suggested within section 4.3 the adapted deep learning model can also be included into established software tools for editing drawings. The embedded model can provide the possibility to import floor plans and their handwritten modifications into a BIM model.

It therefore contributes an important part to automate the process of consistent and concurrent updating of drawings and the corresponding BIM model.

Appendix A

Appendix

Resizing interpolation methods

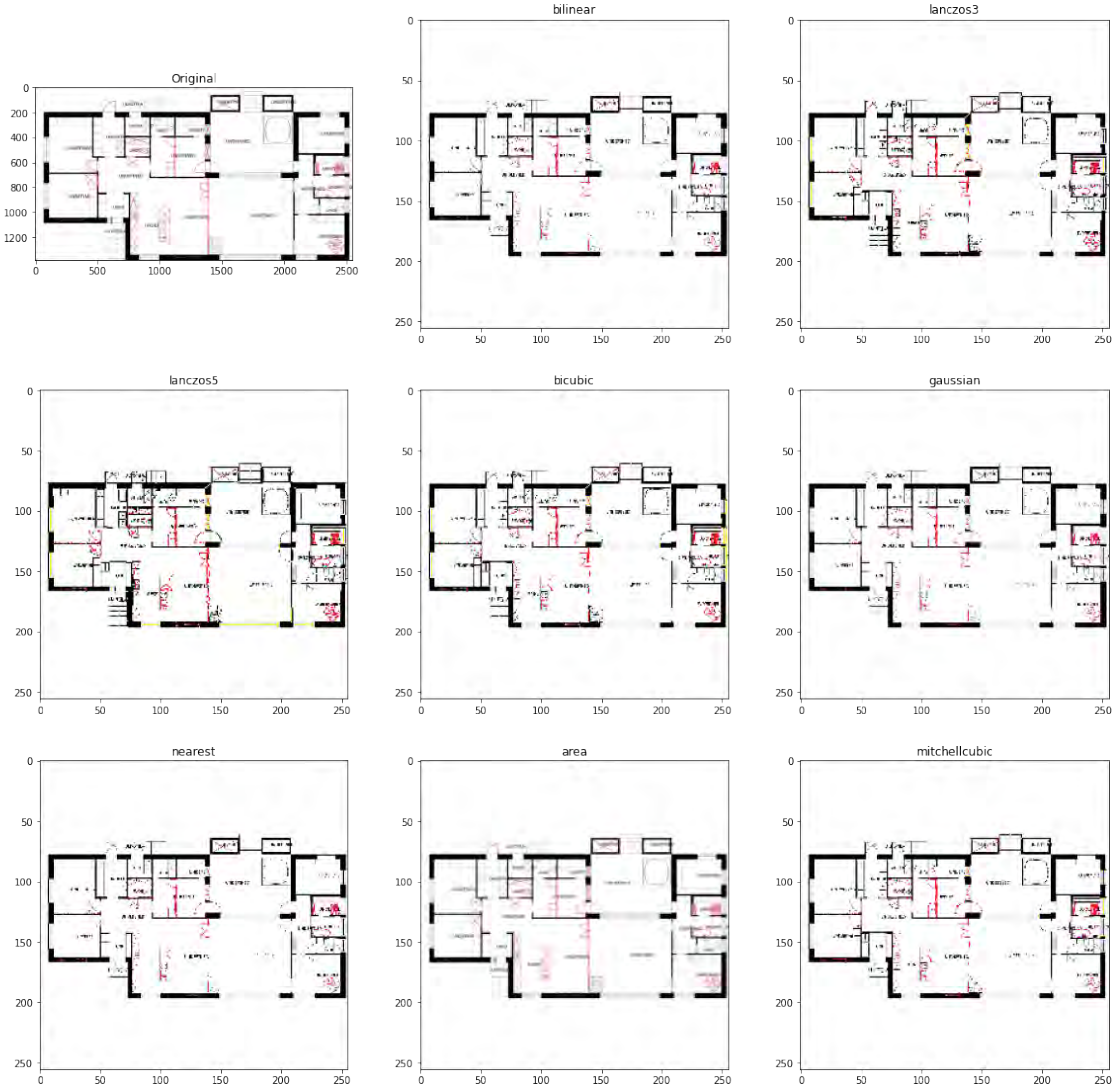


Figure A.1: Comparison of the output of all resizing interpolation methods provided in tensorflow for the biggest image when resizing to 256x256 pixel

Parameter combinations

Table A.1: Training attempts with changing parameter combinations

Parameter combinations for training runs								
#	BATCH	λ	FP size	Epochs	Resize method	Adam Optimiser	Augm.	Purpose
1	1	100	1024	7	nearest	Proposed	0	First Tests
2	1	100	1024	7	nearest	Proposed	1	First Tests
3	1	100	1024	7	nearest	Proposed	2	First Tests
4	1	100	1024	7	nearest	Proposed	7	First Tests
5	1	100	512	7	nearest	Proposed	7	First Tests
6	2	100	512	7	nearest	Proposed	7	First Tests
7	4	100	512	7	nearest	Proposed	7	First Tests
8	1	100	256	7	area	Proposed	7	First Tests
9	2	100	256	60	area	Proposed	7	First Tests
10	4	100	256	120	area	Proposed	7	First Tests
11	8	100	256	240	area	Proposed	7	First Tests
12	2	100	256	120	area	Default	7	First Tests
13	1	1000	256	76	area	Proposed	7	Batch Size
14	2	1000	256	151	area	Proposed	7	Batch Size
15	4	1000	256	151	area	Proposed	7	Batch Size
16	8	1000	256	303	area	Proposed	7	Batch Size
17	16	1000	256	260	area	Proposed	7	Batch Size
18	1	10	512	56	area	Proposed	7	Lambda
19	1	100	512	56	area	Proposed	7	Lambda
20	1	1000	512	56	area	Proposed	7	Lambda
21	1	10000	512	27	area	Proposed	7	Lambda
22	1	None	512	56	area	Proposed	7	Lambda
23	1	1000	512	56	area	$\eta = 1e-5$	7	Optimiser
24	1	1000	512	76	area	$\eta = 1e-4$	7	Optimiser
25	1	1000	512	76	area	$\eta = 2e-4$	7	Optimiser
26	1	1000	512	45	area	$\eta = 1e-3$	7	Optimiser
27	1	1000	512	56	area	Proposed	7	No Shuffle
28	1	1000	512	76	area	Proposed	7	Shuffle
29	1	1000	512	76	area	Proposed	0	Augmentation
30	1	1000	512	76	area	Proposed	7	Augmentation
31	1	1000	512	133	area	$\eta = 1e-4$	7	Training

The results of the first test runs are not stored in the additional results folder. The results and model checkpoints of runs 13 to 31 are stored in the results folder. To identify the fitting results the first two characters of the folder name start with the unique number from this table.

Model architecture

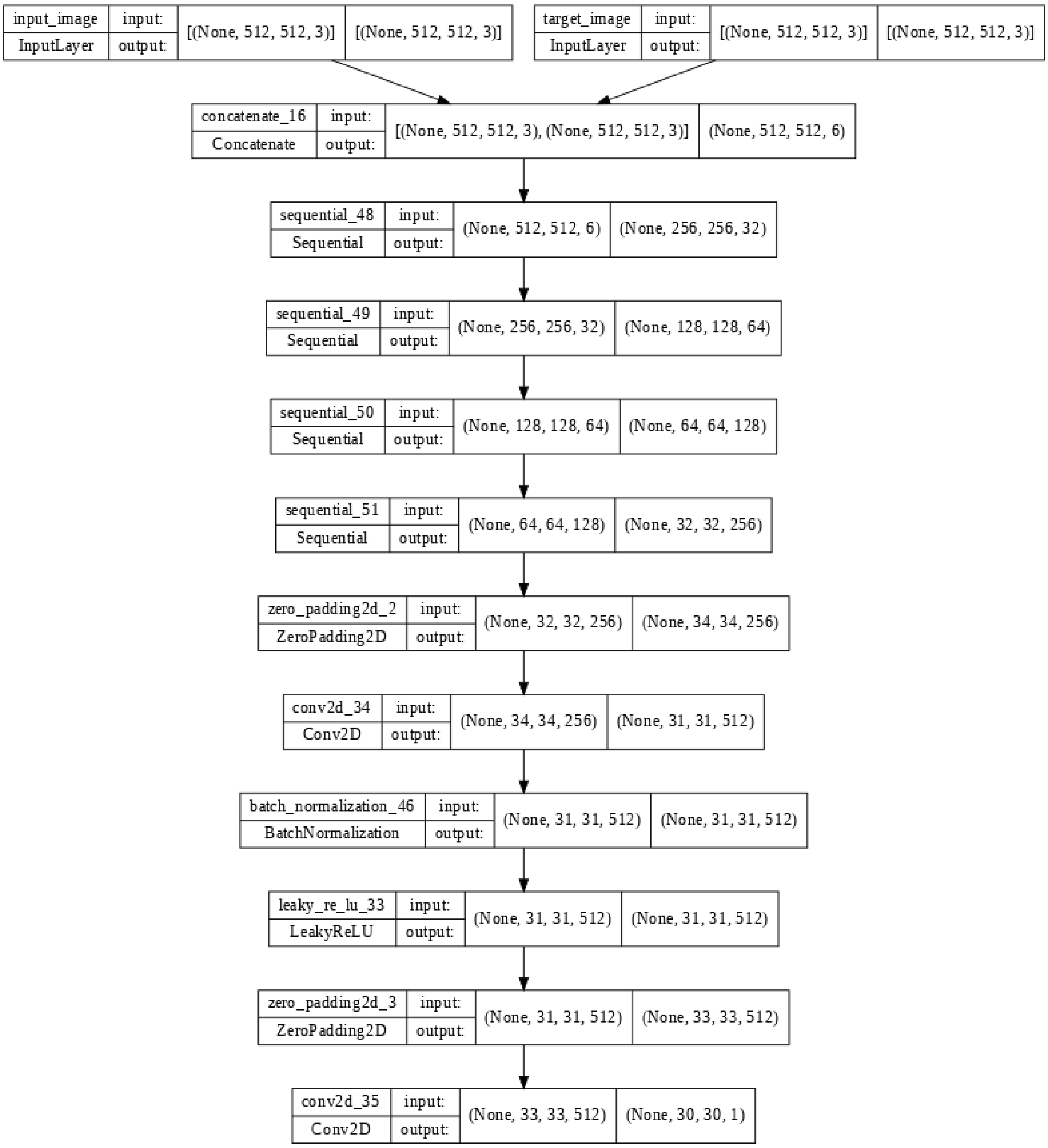

input_image | input: | [(None, 512, 512, 3)] | [(None, 512, 512, 3)]
InputLayer | output:
target_image | input: | [(None, 512, 512, 3)] | [(None, 512, 512, 3)]
InputLayer | output:
concatenate_16 | input: | [(None, 512, 512, 3), (None, 512, 512, 3)] | (None, 512, 512, 6)
Concatenate | output:
sequential_48 | input: | (None, 512, 512, 6) | (None, 256, 256, 32)
Sequential | output:
sequential_49 | input: | (None, 256, 256, 32) | (None, 128, 128, 64)
Sequential | output:
sequential_50 | input: | (None, 128, 128, 64) | (None, 64, 64, 128)
Sequential | output:
sequential_51 | input: | (None, 64, 64, 128) | (None, 32, 32, 256)
Sequential | output:
zero_padding2d_2 | input: | (None, 32, 32, 256) | (None, 34, 34, 256)
ZeroPadding2D | output:
conv2d_34 | input: | (None, 34, 34, 256) | (None, 31, 31, 512)
Conv2D | output:
batch_normalization_46 | input: | (None, 31, 31, 512) | (None, 31, 31, 512)
BatchNormalization | output:
leaky_re_lu_33 | input: | (None, 31, 31, 512) | (None, 31, 31, 512)
LeakyReLU | output:
zero_padding2d_3 | input: | (None, 31, 31, 512) | (None, 33, 33, 512)
ZeroPadding2D | output:
conv2d_35 | input: | (None, 33, 33, 512) | (None, 30, 30, 1)
Conv2D | output:

Figure A.2: Full architecture of the PatchGAN structure of the cGANs Discriminator

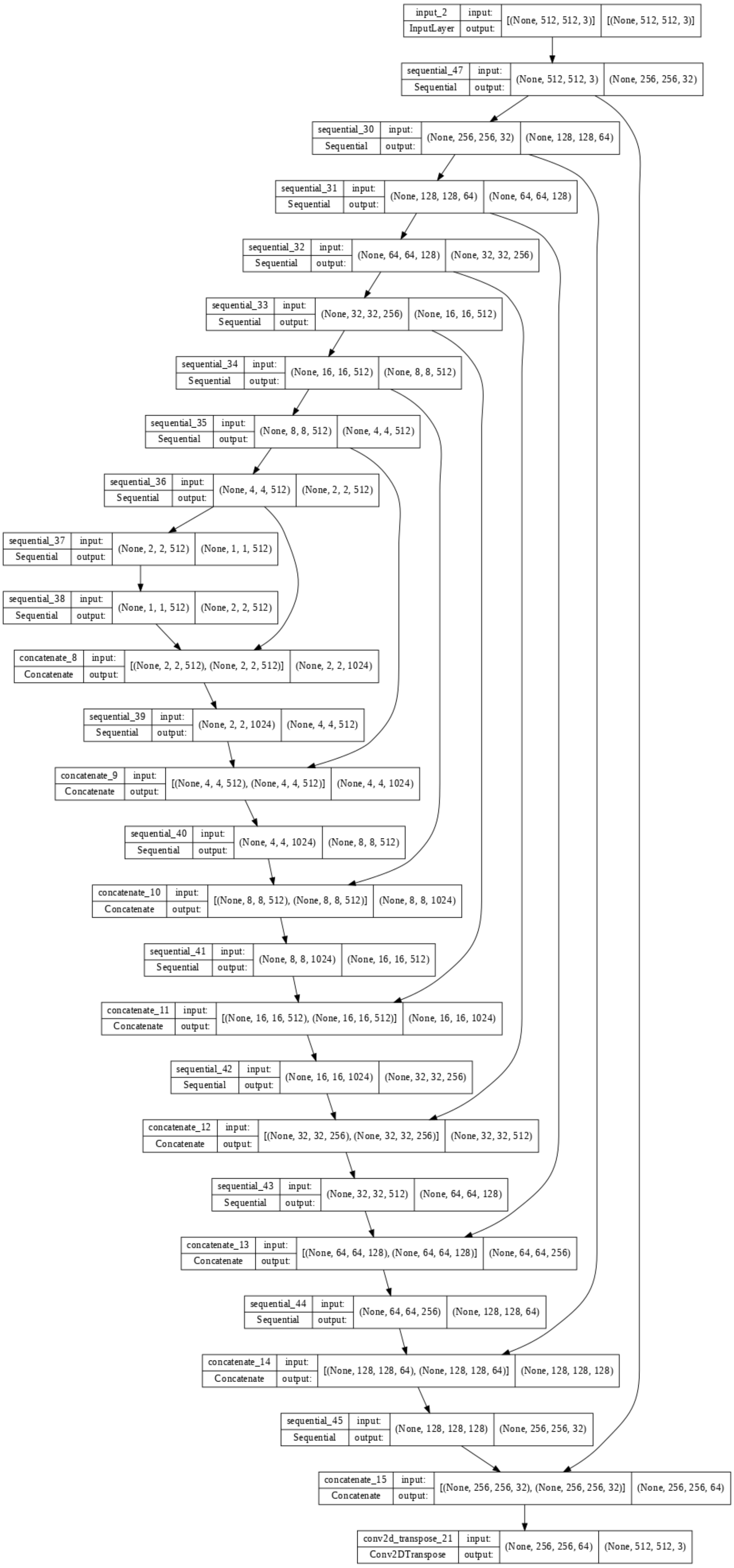

input_2 InputLayer input: output: [(None, 512, 512, 3)] [(None, 512, 512, 3)]
sequential_47 Sequential input: output: (None, 512, 512, 3) (None, 256, 256, 32)
sequential_30 Sequential input: output: (None, 256, 256, 32) (None, 128, 128, 64)
sequential_31 Sequential input: output: (None, 128, 128, 64) (None, 64, 64, 128)
sequential_32 Sequential input: output: (None, 64, 64, 128) (None, 32, 32, 256)
sequential_33 Sequential input: output: (None, 32, 32, 256) (None, 16, 16, 512)
sequential_34 Sequential input: output: (None, 16, 16, 512) (None, 8, 8, 512)
sequential_35 Sequential input: output: (None, 8, 8, 512) (None, 4, 4, 512)
sequential_36 Sequential input: output: (None, 4, 4, 512) (None, 2, 2, 512)
sequential_37 Sequential input: output: (None, 2, 2, 512) (None, 1, 1, 512)
sequential_38 Sequential input: output: (None, 1, 1, 512) (None, 2, 2, 512)
concatenate_8 Concatenate input: output: [(None, 2, 2, 512), (None, 2, 2, 512)] (None, 2, 2, 1024)
sequential_39 Sequential input: output: (None, 2, 2, 1024) (None, 4, 4, 512)
concatenate_9 Concatenate input: output: [(None, 4, 4, 512), (None, 4, 4, 512)] (None, 4, 4, 1024)
sequential_40 Sequential input: output: (None, 4, 4, 1024) (None, 8, 8, 512)
concatenate_10 Concatenate input: output: [(None, 8, 8, 512), (None, 8, 8, 512)] (None, 8, 8, 1024)
sequential_41 Sequential input: output: (None, 8, 8, 1024) (None, 16, 16, 512)
concatenate_11 Concatenate input: output: [(None, 16, 16, 512), (None, 16, 16, 512)] (None, 16, 16, 1024)
sequential_42 Sequential input: output: (None, 16, 16, 1024) (None, 32, 32, 256)
concatenate_12 Concatenate input: output: [(None, 32, 32, 256), (None, 32, 32, 256)] (None, 32, 32, 512)
sequential_43 Sequential input: output: (None, 32, 32, 512) (None, 64, 64, 128)
concatenate_13 Concatenate input: output: [(None, 64, 64, 128), (None, 64, 64, 128)] (None, 64, 64, 256)
sequential_44 Sequential input: output: (None, 64, 64, 256) (None, 128, 128, 64)
concatenate_14 Concatenate input: output: [(None, 128, 128, 64), (None, 128, 128, 64)] (None, 128, 128, 128)
sequential_45 Sequential input: output: (None, 128, 128, 128) (None, 256, 256, 32)
concatenate_15 Concatenate input: output: [(None, 256, 256, 32), (None, 256, 256, 32)] (None, 256, 256, 64)
conv2d_transpose_21 Conv2DTranspose input: output: (None, 256, 256, 64) (None, 512, 512, 3)

Figure A.3: Full architecture of the UNet structure of the cGANs Generator

Original Test Images in size 512x512

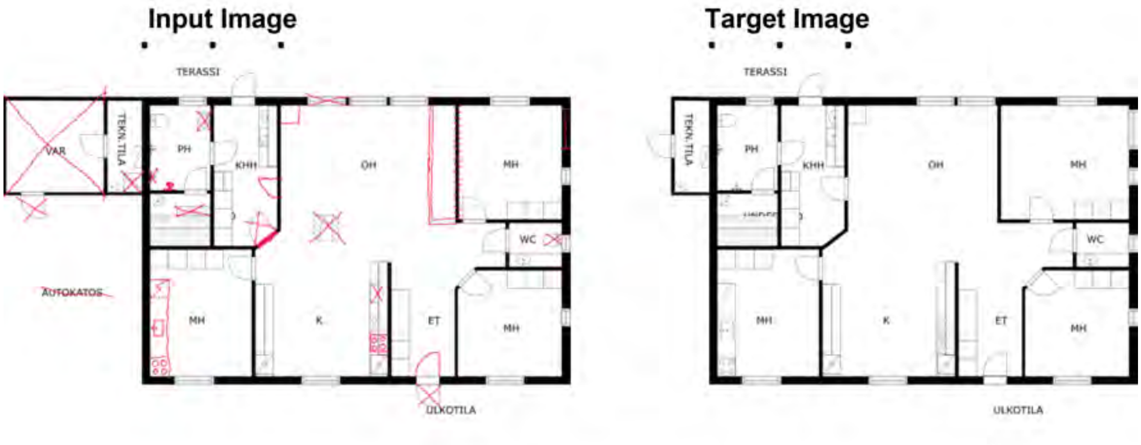


Figure A.4: Input and target image number 1 in a size of 512x512

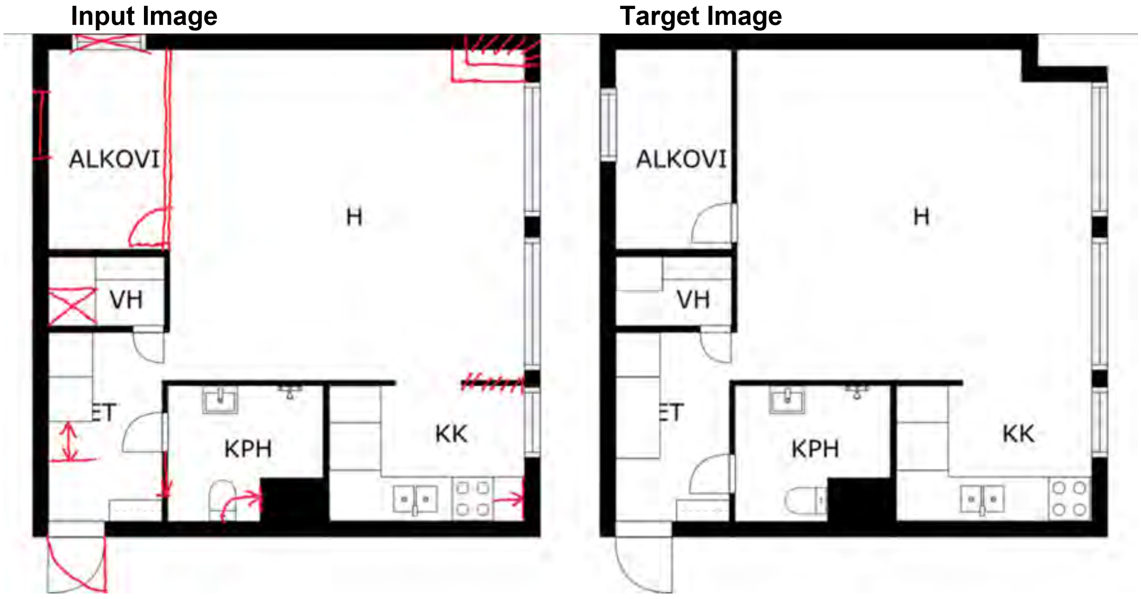


Figure A.5: Input and target image number 2 in a size of 512x512

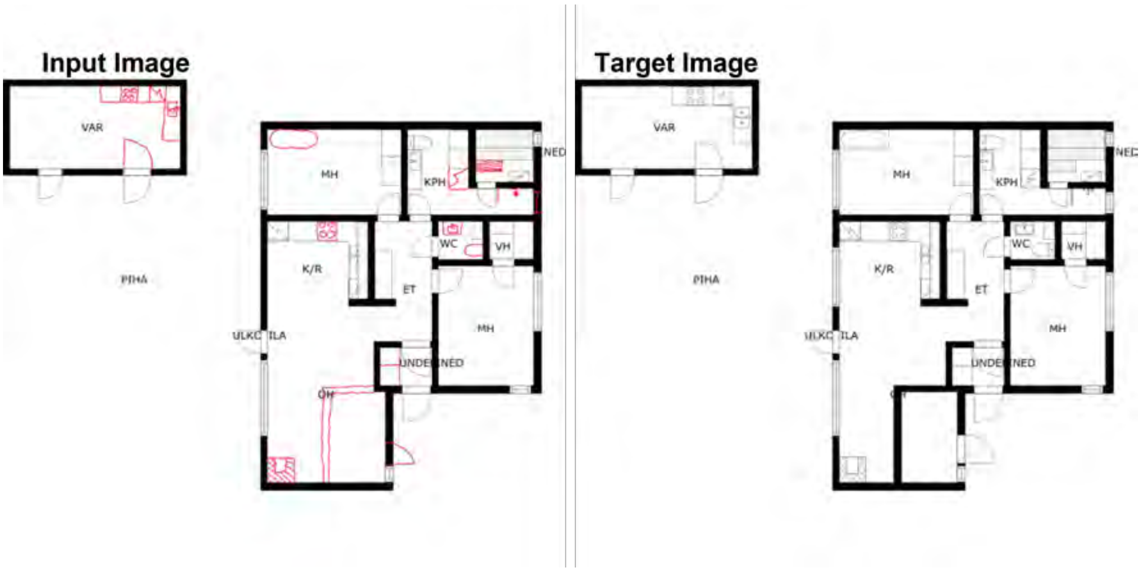


Figure A.6: Input and target image number 3 in a size of 512x512

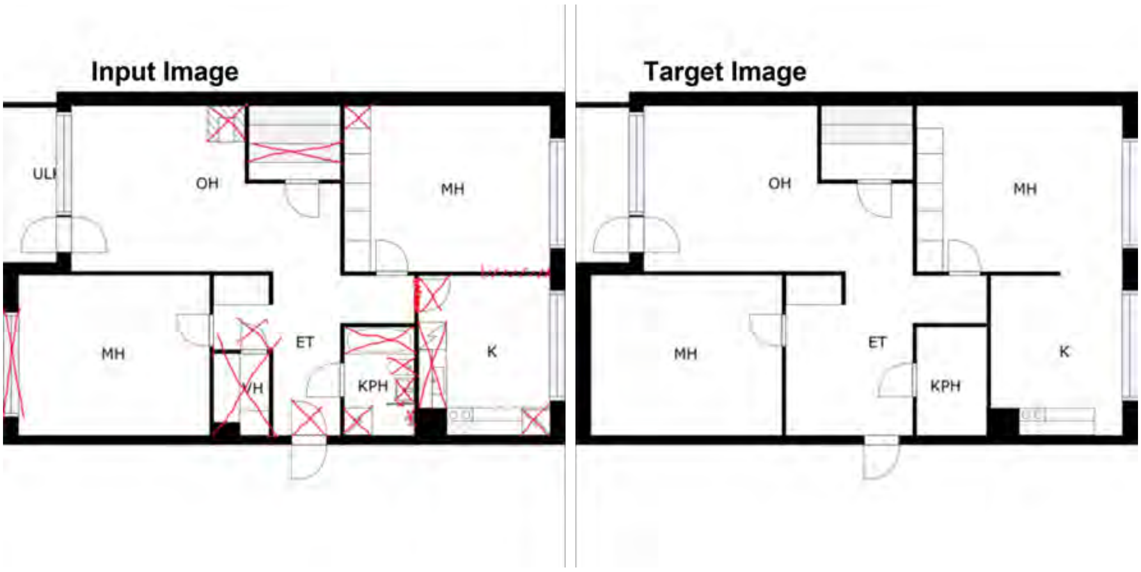


Figure A.7: Input and target image number 4 in a size of 512x512

Resolution Comparison

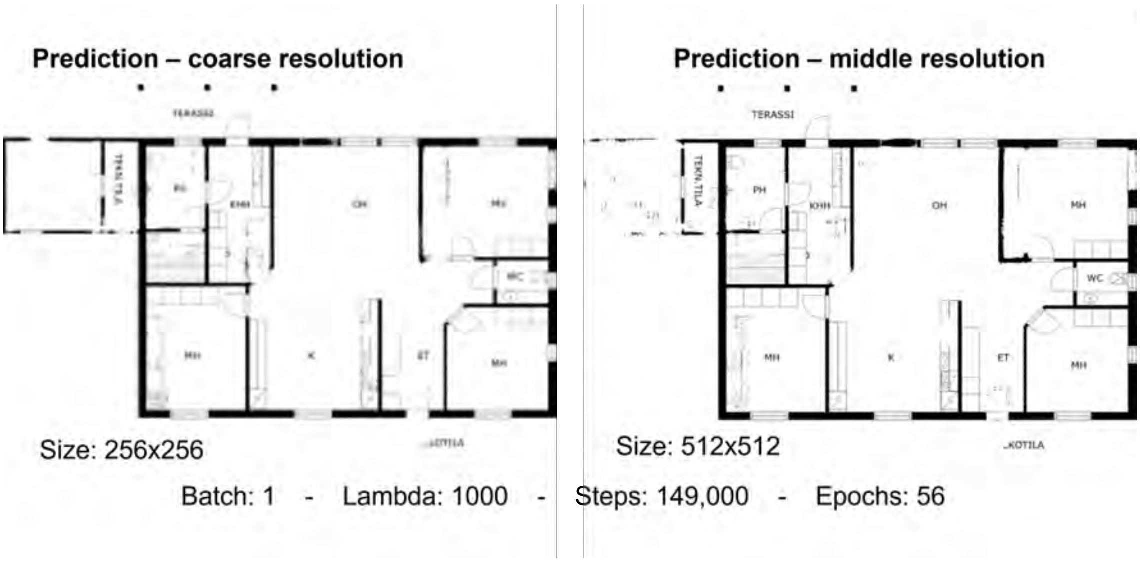

Prediction – coarse resolution
Prediction – middle resolution
TERASSI
TEKN.TILA
PH
KHH
OH
MH
WC
K
ET
MH
KOTILA
Size: 256x256
Size: 512x512
Batch: 1 - Lambda: 1000 - Steps: 149,000 - Epochs: 56

Figure A.8: Comparison of model performance trained on different floor plan resolutions

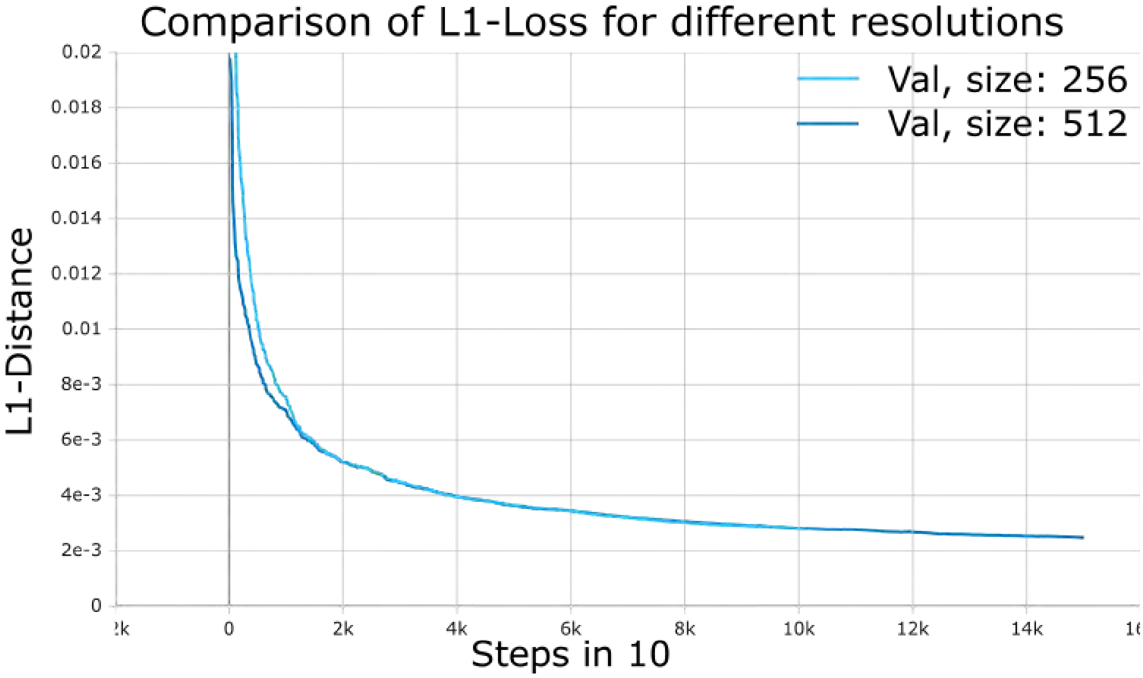
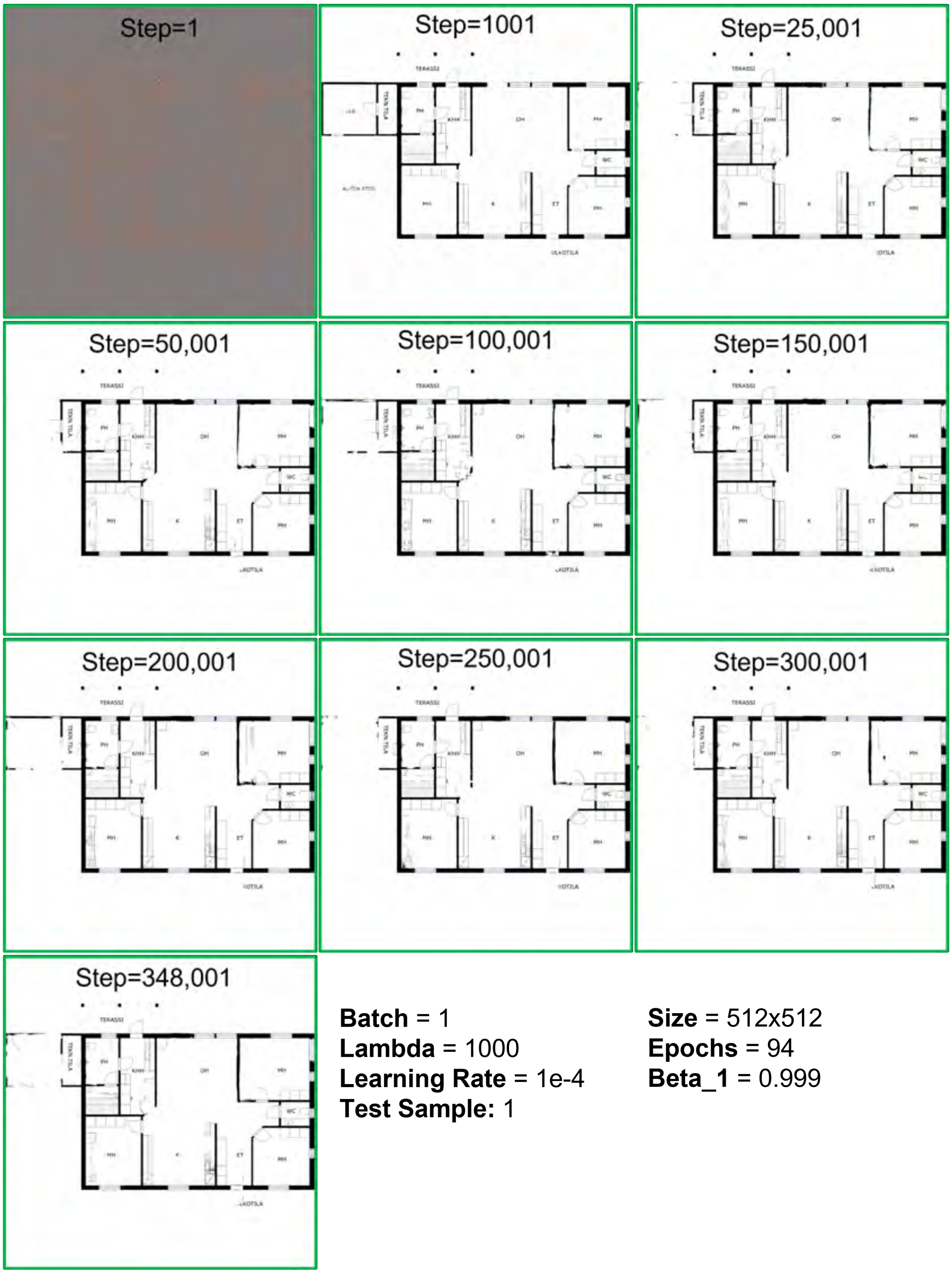


Figure A.9: Difference of validation L1-Loss for changing resolution

Training progress



Batch = 1
Lambda = 1000
Learning Rate = 1e-4
Test Sample: 1

Size = 512x512
Epochs = 94
Beta_1 = 0.999

Figure A.10: Training progress visualised using test sample 1

Comparison of different L1-Loss weights

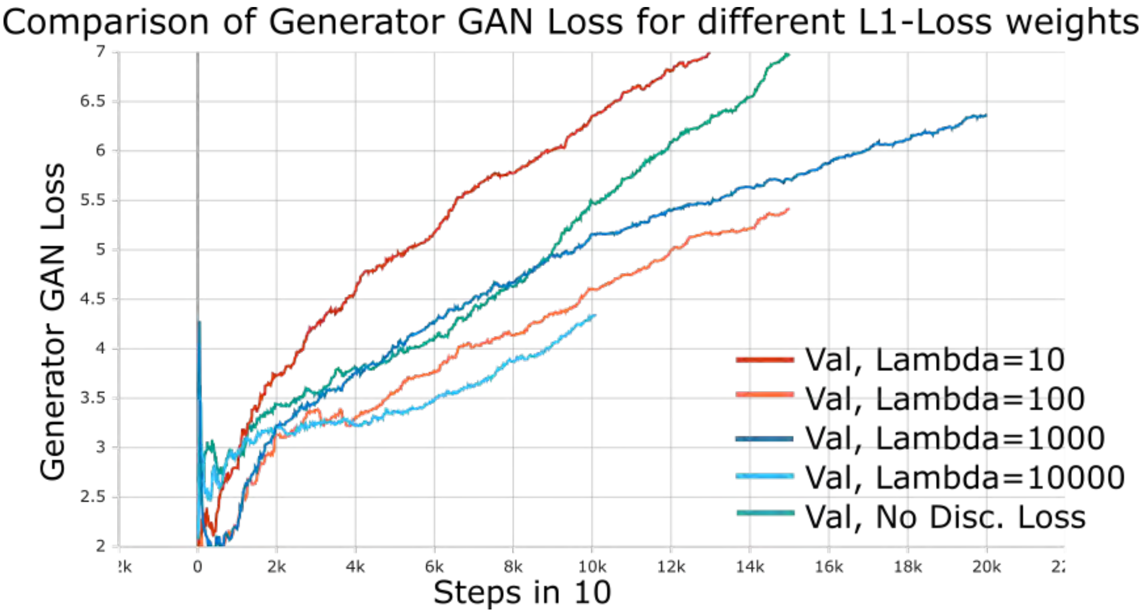


Figure A.11: Difference in the generator GAN validation loss curve for changing the weight of L1-Loss

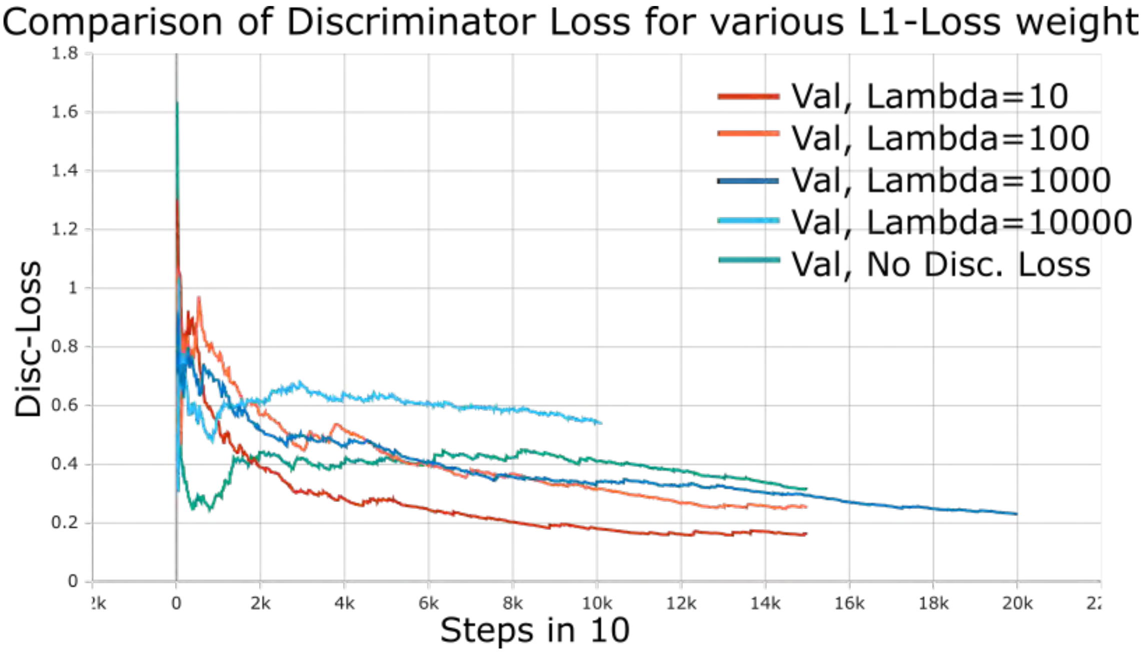


Figure A.12: Difference in the discriminator validation loss curve for changing the weight of L1-Loss

Learning Rate

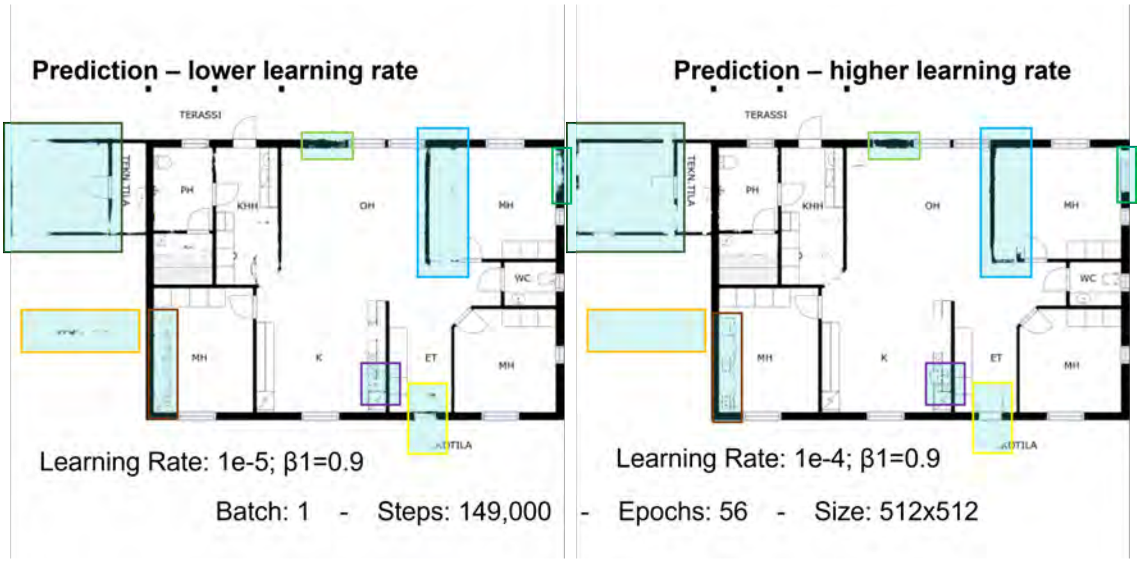

Prediction – lower learning rate
Prediction – higher learning rate
TERASSI
TEKN.TILA
PH
KHH
OH
MH
WC
MH
K
ET
MH
ULKOTILA
Learning Rate: 1e-5; β1=0.9
Learning Rate: 1e-4; β1=0.9
Batch: 1 - Steps: 149,000 - Epochs: 56 - Size: 512x512

Figure A.13: Difference in prediction performance when training with a learning rate of 1e-5 and 1e-4

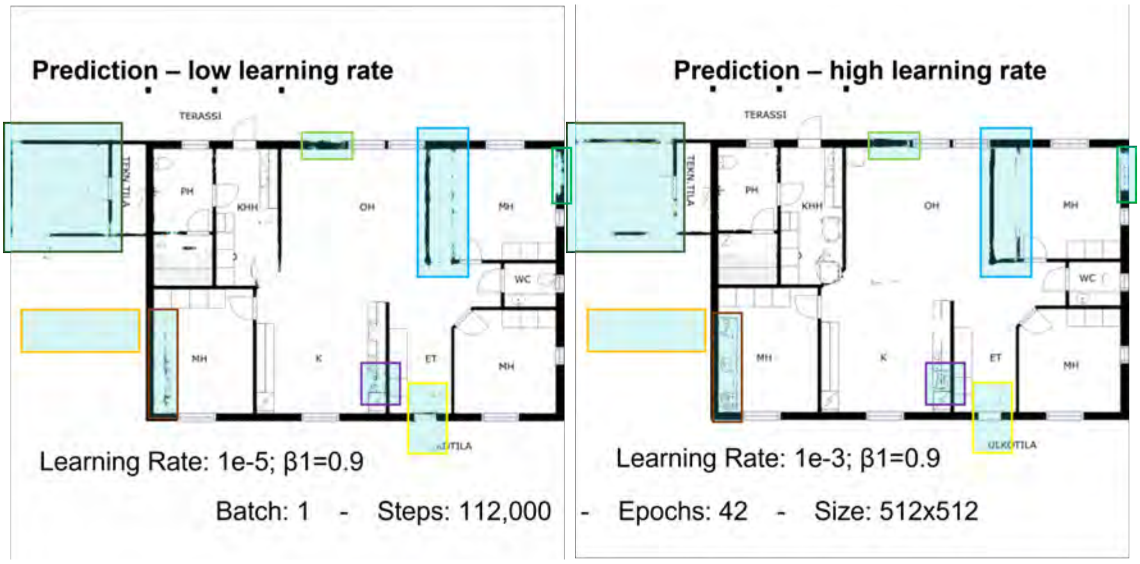

Prediction – low learning rate
Prediction – high learning rate
TERASSI
TEKN.TILA
PH
KHH
OH
MH
WC
MH
K
ET
MH
ULKOTILA
Learning Rate: 1e-5; β1=0.9
Learning Rate: 1e-3; β1=0.9
Batch: 1 - Steps: 112,000 - Epochs: 42 - Size: 512x512

Figure A.14: Difference in prediction performance when training with a learning rate of 1e-5 and 1e-3

Shuffling comparison

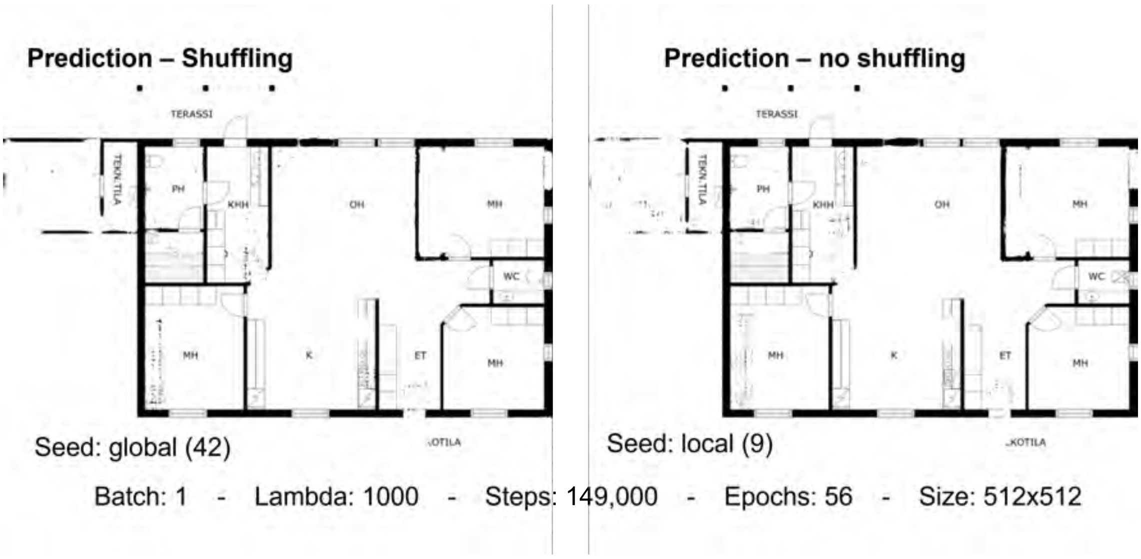

Prediction – Shuffling
Prediction – no shuffling
TERASSI
TEKN.TILA
PH
KHH
OH
MH
WC
MH
K
ET
MH
KOTILA
Seed: global (42)
Seed: local (9)
Batch: 1 - Lambda: 1000 - Steps: 149,000 - Epochs: 56 - Size: 512x512

Figure A.15: Comparison of model performance trained on fixed dataset order and random shuffled order in each epoch

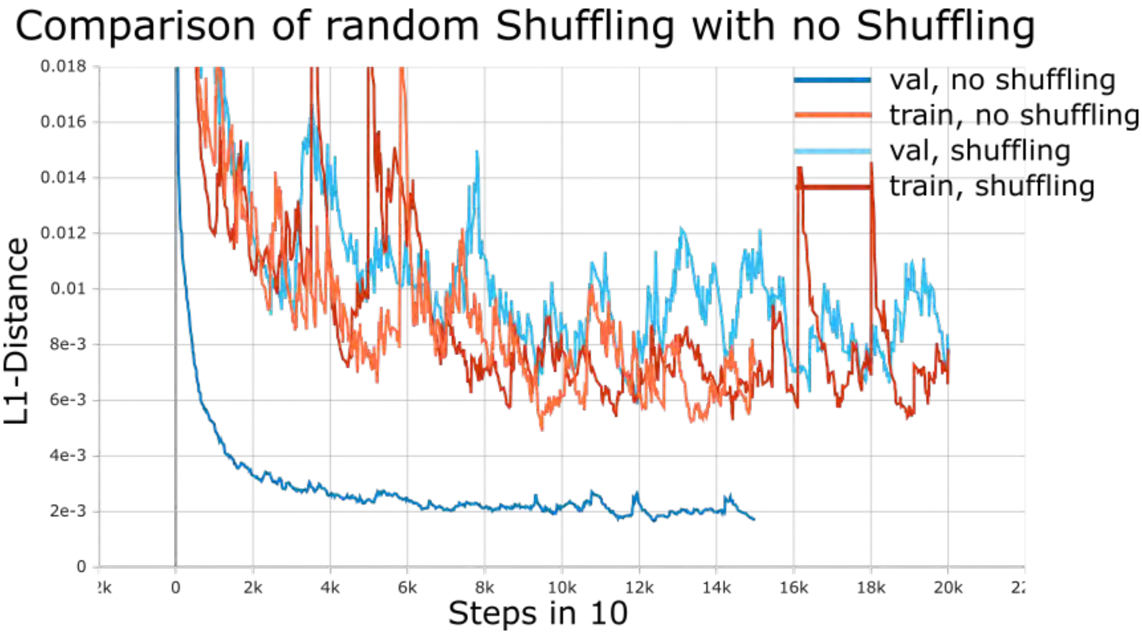
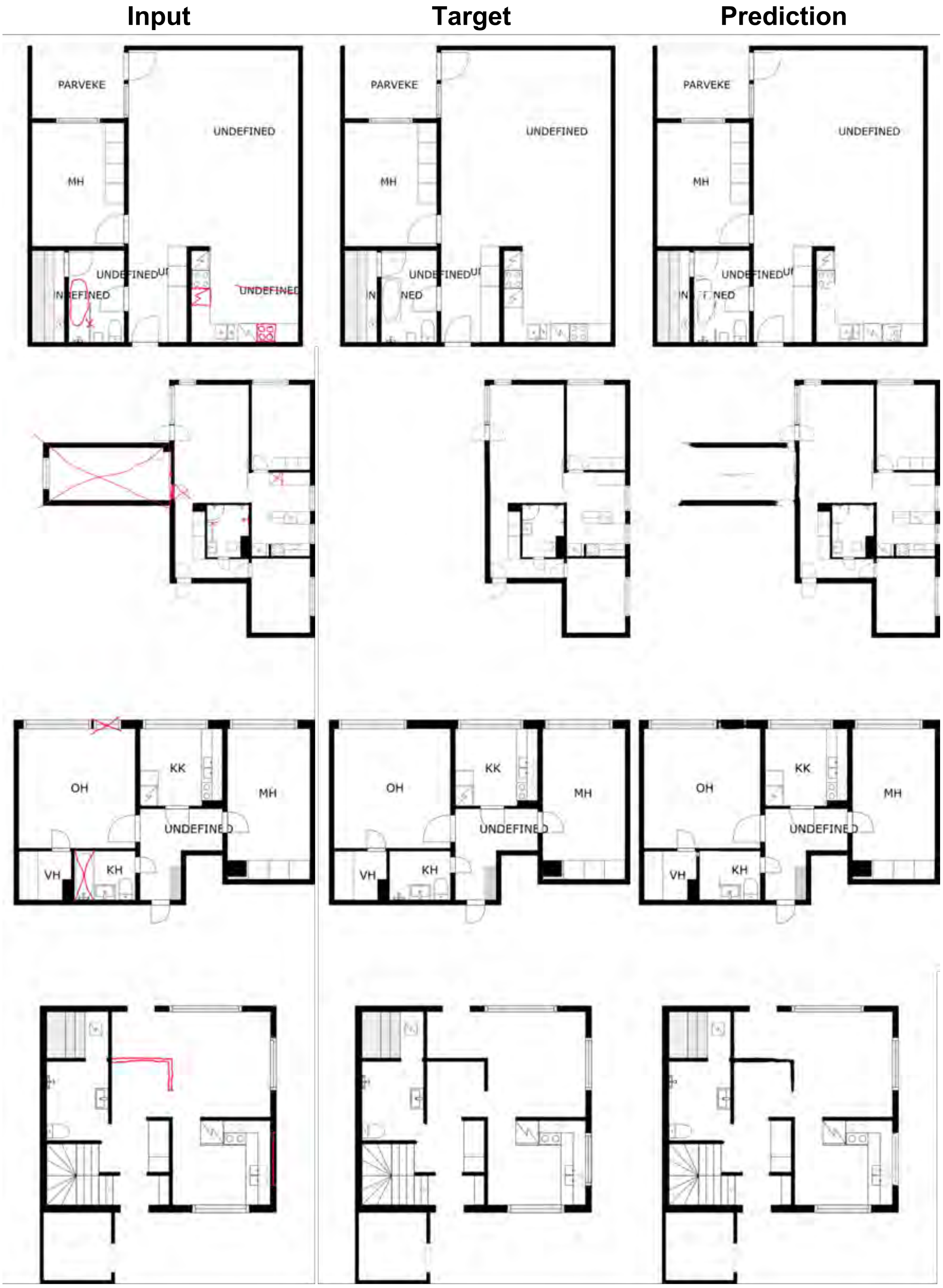


Figure A.16: Difference of training and validation L1-Loss with and without shuffling

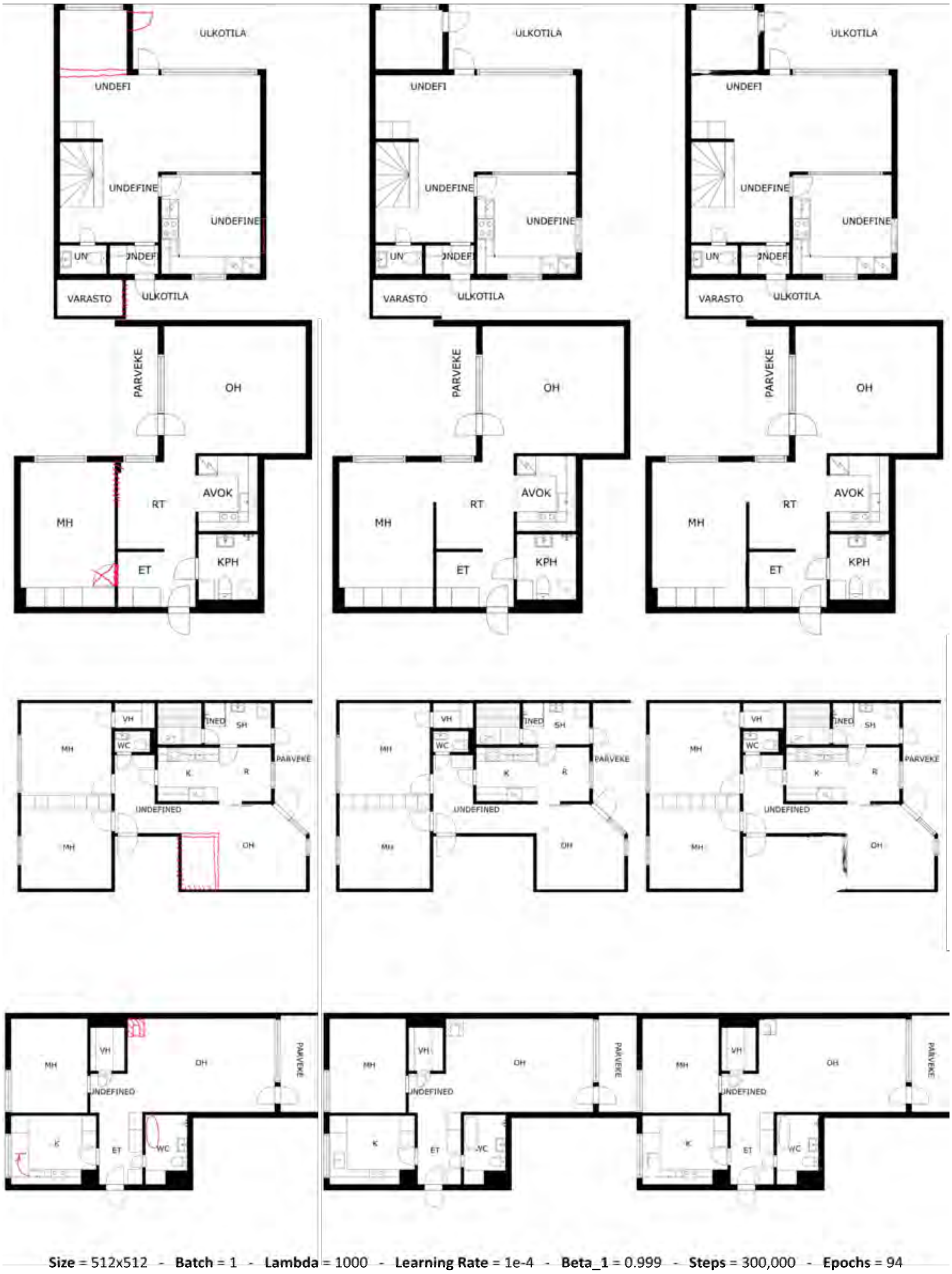
Validation Set - Predictions



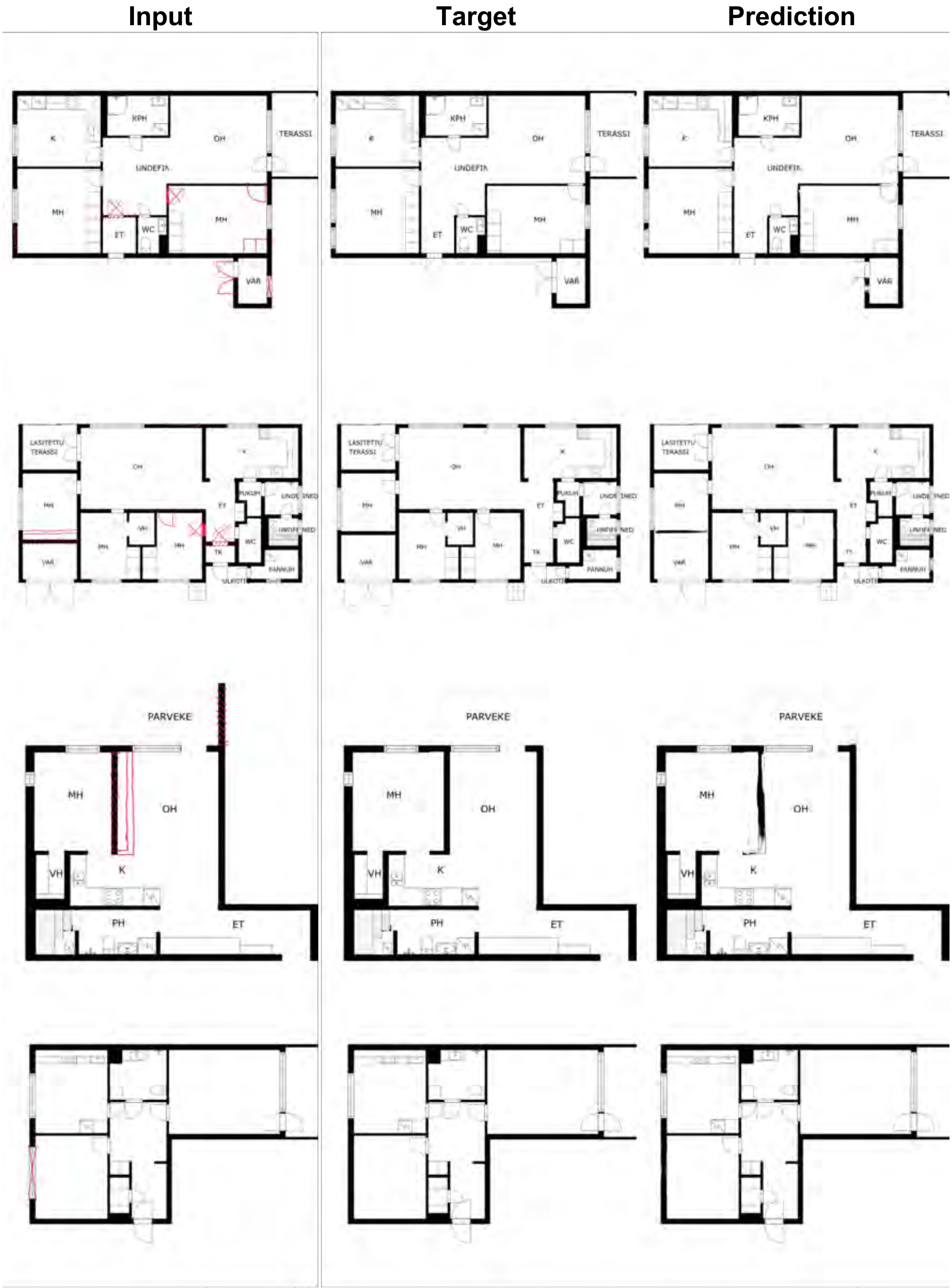
Input Target Prediction

Size = 512x512 - **Batch** = 1 - **Lambda** = 1000 - **Learning Rate** = 1e-4 - **Beta_1** = 0.999 - **Steps** = 300,000 - **Epochs** = 94

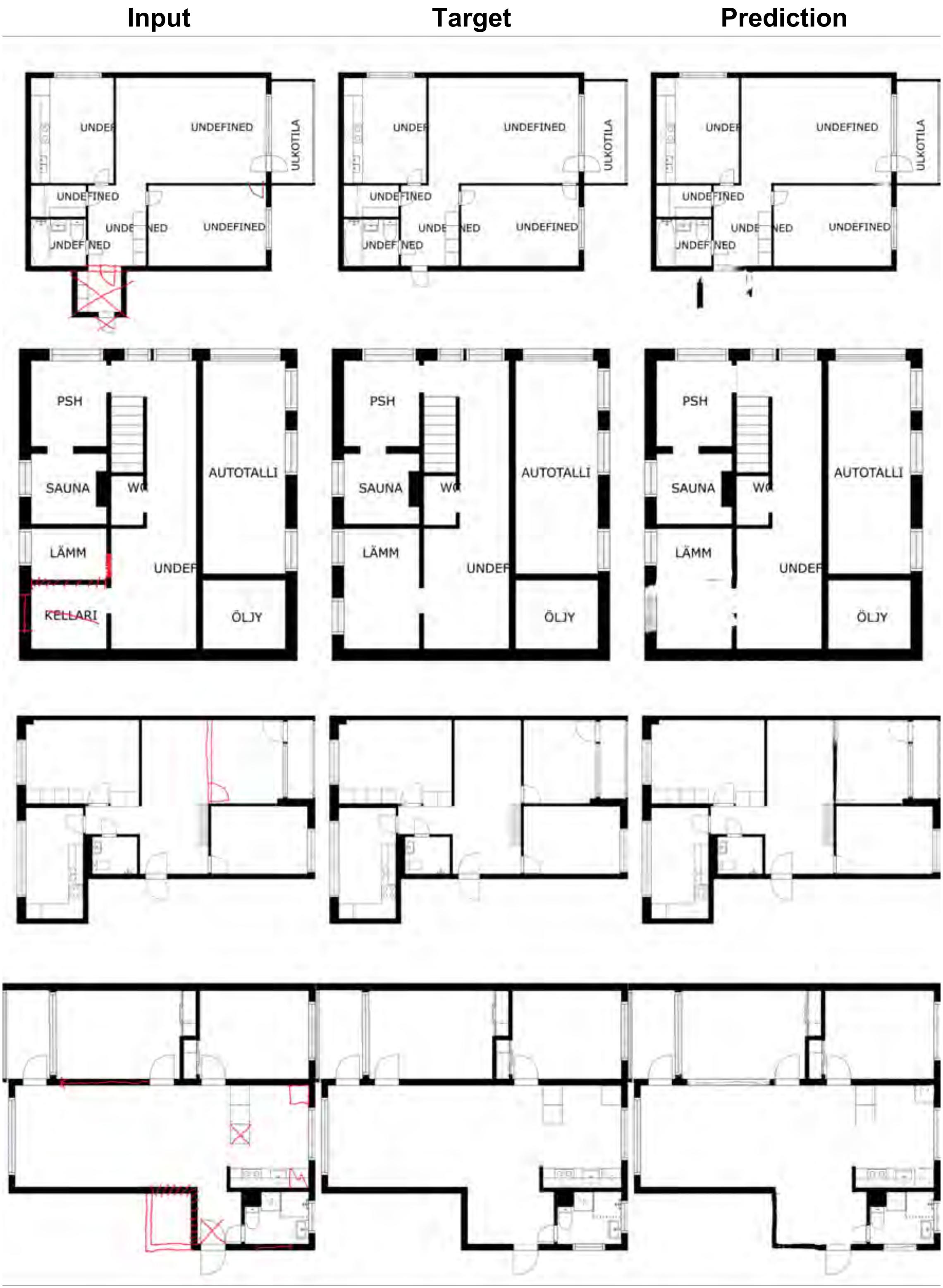
Input Target Prediction



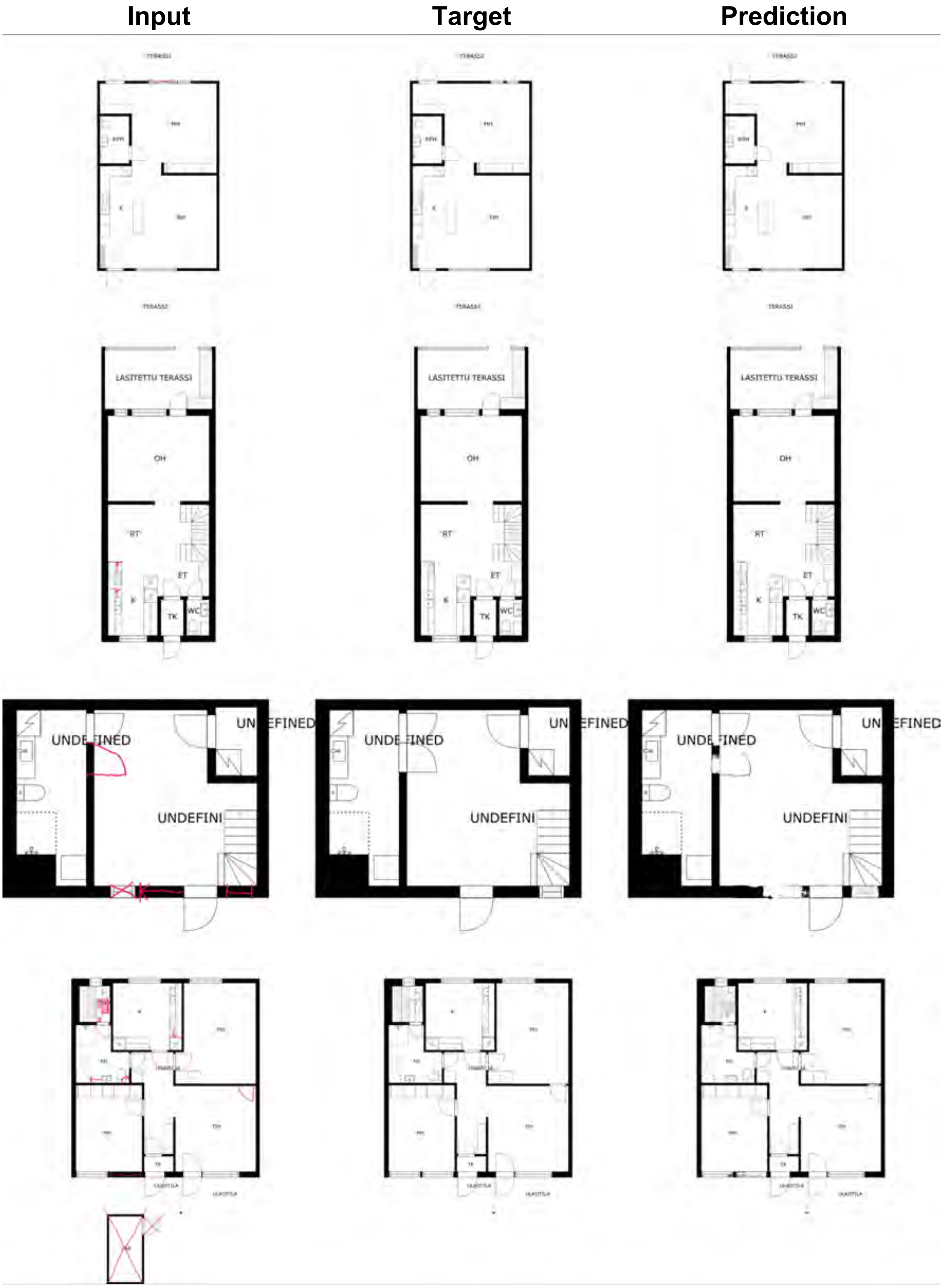
Size = 512x512 - Batch = 1 - Lambda = 1000 - Learning Rate = 1e-4 - Beta_1 = 0.999 - Steps = 300,000 - Epochs = 94



Size = 512x512 - **Batch** = 1 - **Lambda** = 1000 - **Learning Rate** = 1e-4 - **Beta_1** = 0.999 - **Steps** = 300,000 - **Epochs** = 94



Size = 512x512 - **Batch** = 1 - **Lambda** = 1000 - **Learning Rate** = 1e-4 - **Beta_1** = 0.999 - **Steps** = 300,000 - **Epochs** = 94


Input

Target

Prediction

Size = 512x512 - **Batch** = 1 - **Lambda** = 1000 - **Learning Rate** = 1e-4 - **Beta_1** = 0.999 - **Steps** = 300,000 - **Epochs** = 94

References

ABUALDENIEN, J., & BORRMANN, A. (2022). Ensemble-learning approach for the classification of levels of geometry (log) of building elements. *Advanced Engineering Informatics*, *51*, 101497. https://doi.org/10.1016/j.aei.2021.101497

ADALOGLOU, N. (2020). Intuitive explanation of skip connections in deep learning. *https://theaisummer.com*. https://theaisummer.com/skip-connections/

AGGARWAL, C. C. (2018). An introduction to neural networks. *Neural networks and deep learning: A textbook* (pp. 1–52). Springer International Publishing. https://doi.org/10.1007/978-3-319-94463-0{\textunderscore}1

AHTI KALERVO, JUHA YLIOINAS, MARKUS HÄIKIÖ, ANTTI KARHU, & JUHO KANNALA. (2019). Cubicasa5k. https://doi.org/10.5281/zenodo.2613548

ALZUBAIDI, L., ZHANG, J., HUMAIDI, A. J., AL-DUJAILI, A., DUAN, Y., AL-SHAMMA, O., SANTAMARÍA, J., FADHEL, M. A., AL-AMIDIE, M., & FARHAN, L. (2021). Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), 53. https://doi.org/10.1186/s40537-021-00444-8

BHALLEY, R. (2021a). Computer vision. In R. BHALLEY (Ed.), *Deep learning with swift for tensorflow: Differentiable programming with swift* (pp. 238–267). Apress.

BHALLEY, R. (2021b). Neural networks. In R. BHALLEY (Ed.), *Deep learning with swift for tensorflow: Differentiable programming with swift* (pp. 171–230). Apress. https://doi.org/10.1007/978-1-4842-6330-3{\textunderscore}5

BORRMANN, A., ABUALDENIEN, J., & KRIJNEN, T. (2021). Information containers providing deep linkage of drawings and bim models. *Proc. of the CIB W78 Conference 2021*.

BOTTOU, L. (2012). Stochastic gradient descent tricks. In G. MONTAVON, G. B. ORR, & K.-R. MÜLLER (Eds.), *Neural networks: Tricks of the trade: Second edition* (pp. 421–436). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8{\textunderscore}25

BROWNLEE, J. (2015). Basic concepts in machine learning. https://machinelearningmastery.com/basic-concepts-in-machine-learning/

BUCKLEY, B. C. (2012). Model-based learning. In N. M. SEEL (Ed.), *Encyclopedia of the sciences of learning* (pp. 2300–2303). Springer US. https://doi.org/10.1007/978-1-4419-1428-6{\textunderscore}589

CHOLLET, F. et al. (2015). Keras. https://github.com/fchollet/keras

CHRISTINA KOURIDI. (2019). Vanilla gan with numpy. https://christinakouridi.blog/2019/07/09/vanilla-gan-numpy/

CIOS, K. J., SWINIARSKI, R. W., PEDRYCZ, W., & KURGAN, L. A. (2007). Unsupervised learning: Association rules. *Data mining: A knowledge discovery approach* (pp. 289–306). Springer US. https://doi.org/10.1007/978-0-387-36795-8{\textunderscore}10

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, *35*(1), 53–65.

Cruz, S., Hutchcroft, W., Li, Y., Khosravan, N., Boyadzhiev, I., & Kang, S. B. (2021). Zillow indoor dataset: Annotated floor plans with 360º panoramas and 3d room layouts. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2133–2143.

Da Silva, T. A. (n.d.). Dcgan hyperparameter tuning - part 2. *Sdet - object recognition at magic leap*. https://www.linkedin.com/pulse/dcgan-hyperparameter-tuning-part-2-thiago-abreu-da-silva

Dabbura, I. (2017). Gradient descent algorithm and its variants. https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Delalandre, M. (2014). Sesyd. http://mathieu.delalandre.free.fr/projects/sesyd/index.html

de las Heras, L., Terrades, O., Robles, S., & S'anchez, G. (2015). Cvc-fp and sgt: A new database for structural floor plan analysis and its groundtruthing tool. *International Journal on Document Analysis and Recognition*.

de Wilde, P. (1996). Backpropagation. In P. de Wilde (Ed.), *Neural networks models: An analysis* (pp. 35–51). Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0034480

Dodge, S., Xu, J., & Stenger, B. (2017). *Parsing floor plan images*. https://doi.org/10.23919/MVA.2017.7986875

Dong, S., Wang, W., Li, W., & Zou, K. (2021). Vectorization of floor plans based on edgegan. *Information*, *12*, 206. https://doi.org/10.3390/info12050206

Fan, Z., Zhu, L., Li, H., Zhu, S., & Tan, P. (2021). Floorplancad: A large-scale cad drawing dataset for panoptic symbol. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

Feng, J., He, X., Teng, Q., Ren, C., Chen, H., & Li, Y. (2019). Reconstruction of porous media from extremely limited information using conditional generative adversarial networks. *Physical Review E*, *100*. https://doi.org/10.1103/PhysRevE.100.033308

Fradkov, A. L. (2020). Early history of machine learning. *IFAC-PapersOnLine*, *53*(2), 1385–1390. https://doi.org/10.1016/j.ifacol.2020.12.1888

Frajberg, D. (10/30/2017). Introduction to the artificial intelligence and computer vision revolution. https://www.slideshare.net/darian_f/introduction-to-the-artificial-intelligence-and-computer-vision-revolution

Gallant, S. I. (1993). *Neural network learning and expert systems*. MIT Press. https://books.google.de/books?id=N8i6pTafq1kC

Ganokratanaa, T., Aramvith, S., & Sebe, N. (2020). Unsupervised anomaly detection and localization based on deep spatiotemporal translation network. *IEEE Access*, *PP*, 1. https://doi.org/10.1109/ACCESS.2020.2979869

Garbin, C., Zhu, X., & Marques, O. (2020). Dropout vs. batch normalization: An empirical study of their impact to deep learning. *Multimedia Tools and Applications*, *79*(19), 12777–12815. https://doi.org/10.1007/s11042-019-08453-9

Géron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems / aurélien géron* (Second Edition). O'Reilly.

Gómez, R. (n.d.). Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names. https://gombru.github.io/2018/05/23/cross_entropy_loss/

Goyal, S., Mistry, V., Chattopadhyay, C., & Bhatnagar, G. (2019). Bridge: Building plan repository for image description generation, and evaluation. *2019 International Conference on Document Analysis and Recognition (ICDAR)*, 1071–1076. https://doi.org/10.1109/ICDAR.2019.00174

Gupta, K. (2021). Hidden layers: Machine learning (ml). https://iq.opengenus.org/hidden-layers/

Haibing Wu, & Xiaodong Gu. (2015). Towards dropout training for convolutional neural networks. *CoRR*, *abs/1512.00242*.

Inkscape Project. (2022). Inkscape (1.1.2). https://inkscape.org

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-image translation with conditional adversarial networks. https://doi.org/10.48550/ARXIV.1611.07004

Jakhar, D., & Kaur, I. (2020). Artificial intelligence, machine learning and deep learning: Definitions and differences. *Clinical and Experimental Dermatology*, *45*(1), 131–132. https://doi.org/10.1111/ced.14029

Junho Jo, Hyung Il Koo, Jae Woong Soh, & Nam Ik Cho. (2019). Handwritten text segmentation via end-to-end learning of convolutional neural network. https://arxiv.org/abs/1906.05229

Kalervo, A., Ylioinas, J., Häikiö, M., Karhu, A., & Kannala, J. (2019). Cubicasa5k: A dataset and an improved multi-task model for floorplan image analysis. *Image Analysis: 21st Scandinavian Conference, SCIA 2019, Norrköping, Sweden, June 11–13, 2019, Proceedings*, 28–40. https://doi.org/10.1007/978-3-030-20205-7{\textunderscore}3

Kandel, I., & Castelli, M. (2020). The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT Express*, *6*(4), 312–315. https://doi.org/10.1016/j.icte.2020.04.010

Kim, S. E., & Seo, I. (2015). Artificial neural network ensemble modeling with conjunctive data clustering for water quality prediction in rivers. *Journal of Hydro-environment Research*, *9*. https://doi.org/10.1016/j.jher.2014.09.006

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Labach, A., Salehinejad, H., & Valaee, S. (2019). Survey of dropout methods for deep neural networks. https://doi.org/10.48550/ARXIV.1904.13310

Legg, S., & Hutter, M. (2007). Universal intelligence: A definition of machine intelligence. *Minds and Machines*, *17*(4), 391–444. https://doi.org/10.1007/s11023-007-9079-x

LI, Y., WANG, Q., ZHANG, J., HU, L., & OUYANG, W. (2021). The theoretical research of generative adversarial networks: An overview. *Neurocomputing*, *435*, 26–41. https://doi.org/10.1016/j.neucom.2020.12.114

LIU, C., WU, J., & FURUKAWA, Y. (2018). Floornet: A unified framework for floorplan reconstruction from 3d scans. *CoRR*, *abs/1804.00090*.

LORENZ, U. (2020). Bestärkendes lernen als teilgebiet des maschinellen lernens. *Reinforcement learning: Aktuelle ansätze verstehen - mit beispielen in java und greenfoot* (pp. 1–11). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-61651-2{\textunderscore}1

MARTIN ABADI, ASHISH AGARWAL, PAUL BARHAM, EUGENE BREVDO, ZHIFENG CHEN, CRAIG CITRO, GREG S. CORRADO, ANDY DAVIS, JEFFREY DEAN, MATTHIEU DEVIN, SANJAY GHEMAWAT, IAN GOODFELLOW, ANDREW HARP, GEOFFREY IRVING, MICHAEL ISARD, YANGQING JIA, RAFAL JOZEFOWICZ, LUKASZ KAISER, MANJUNATH KUDLUR, . . . XIAOQIANG ZHENG. (2015). Tensorflow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/

MIGILINSKAS, D., POPOV, V., JUOCEVICIUS, V., & USTINOVICHIUS, L. (2013). The benefits, obstacles and problems of practical bim implementation. *Procedia Engineering*, *57*, 767–774. https://doi.org/10.1016/j.proeng.2013.04.097

MIRZA, M., & OSINDERO, S. (2014). Conditional generative adversarial nets. https://doi.org/10.48550/ARXIV.1411.1784

MISHRA, S., HASHMI, K. A., PAGANI, A., LIWICKI, M., STRICKER, D., & AFZAL, M. Z. (2021). Towards robust object detection in floor plan images: A data augmentation approach. *Applied Sciences*, *11*, 11174. https://doi.org/10.3390/app112311174

MUNRO, P. (2017). Backpropagation. In C. SAMMUT & G. I. WEBB (Eds.), *Encyclopedia of machine learning and data mining* (pp. 93–97). Springer US. https://doi.org/10.1007/978-1-4899-7687-1{\textunderscore}51

PANT, A. (2019). Workflow of a machine learning project. *towards data science*.

PAPER, D. (2018). Gradient descent. *Data science fundamentals for python and mongodb* (pp. 97–128). Apress. https://doi.org/10.1007/978-1-4842-3597-3{\textunderscore}4

QIUHONG KE, JUN LIU, MOHAMMED BENNAMOUN, SENJIAN AN, FERDOUS SOHEL, & FARID BOUSSAID. (2018). Chapter 5 - computer vision for human–machine interaction. In MARCO LEO & GIOVANNI MARIA FARINELLA (Eds.), *Computer vision for assistive healthcare* (pp. 127–145). Academic Press. https://doi.org/10.1016/B978-0-12-813445-0.00005-8

RAKUTEN INSTITUTE OF TECHNOLOGY. (2017). Rakuten data release. https://rit.rakuten.com/data_release/

REZVANIFAR, A., COTE, M., & BRANZAN ALBU, A. (2019). Symbol spotting for architectural drawings: State-of-the-art and new industry-driven developments. *IPSJ Transactions on Computer Vision and Applications*, *11*(1), 2. https://doi.org/10.1186/s41074-019-0055-1

RONNEBERGER, O., FISCHER, P., & BROX, T. (2015). U-net: Convolutional networks for biomedical image segmentation. https://doi.org/10.48550/ARXIV.1505.04597

Ruder, S. (2016). An overview of gradient descent optimization algorithms.

Sarker, I. H. (2021). Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, *2*(3), 160. https://doi.org/10.1007/s42979-021-00592-x

Saxena, D., & Cao, J. (2020). *Generative adversarial networks (gans): Challenges, solutions, and future directions*.

Semenov, A., Boginski, V., & Pasiliao, E. (2019). Neural networks with multidimensional cross-entropy loss functions. https://doi.org/10.1007/978-3-030-34980-6{\textunderscore}5

Sharma, D., Gupta, N., Chattopadhyay, C., & Mehta, S. (2017). Daniel: A deep architecture for automatic analysis and retrieval of building floor plans. *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, *01*, 420–425. https://doi.org/10.1109/ICDAR.2017.76

Sharma, S. (2017). Activation functions in neural networks: Sigmoid, tanh, softmax, relu, leaky relu explained. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Shi, J., Dang, J., Cui, M., Zuo, R., Shimizu, K., Tsunoda, A., & Suzuki, Y. (2021). Improvement of damage segmentation based on pixel-level data balance using vgg-unet. *Applied Sciences*, *11*, pp.518.1–17. https://doi.org/10.3390/app11020518

Srivastava, T. (2014). How does artificial neural network (ann) algorithm work? simplified! https://www.analyticsvidhya.com/blog/2014/10/ann-work-simplified/

Stewart, M. (2019). Simple introduction to convolutional neural networks. https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

Swaminathan, A. (2020). Summer school series: Lecture 5: Deep learning in computer vision. https://archana1998.github.io/post/rahul-sukthankar/

Tambe, A. (2020). Types of ml systems. https://medium.com/@aj.tambe.02/types-of-ml-systems-160601843758

Trzeciak, M., & Borrmann, A. (2018). Towards registration of construction drawings to building information models using knowledge-based extended geometric hashing. *EG-ICE*.

Wang, K., Gou, C., Duan, Y., Yilun, L., Zheng, X., & Wang, F.-Y. (2017). Generative adversarial networks: Introduction and outlook. *4*, 588–598. https://doi.org/10.1109/JAS.2017.7510583

Yingge, H., Ali, I., & Lee, K.-Y. (2020). *Deep neural networks on chip - a survey*. https://doi.org/10.1109/BigComp48618.2020.00016

Zeng, Z., Li, X., Yu, Y. K., & Fu, C.-W. (2019). Deep floor plan recognition using a multi-task network with room-boundary-guided attention. *CoRR*, *abs/1908.11025*.

Zheng, S., Lu, J., Ghasemzadeh, N., Hayek, S., Quyyumi, A., & Wang, F. (2017). Effective information extraction framework for heterogeneous clinical reports using online machine learning and controlled vocabularies. *JMIR Medical Informatics*, *5*, e12. https://doi.org/10.2196/medinform.7235

Ziran, Z., & Marinai, S. (2018). Object detection in floor plan images. In L. Pancioni, F. Schwenker, & E. Trentin (Eds.), *Artificial neural networks in pattern recognition* (pp. 383–394). Springer International Publishing.

Declaration

I hereby affirm that I have independently written the thesis submitted by me and have not used any sources or aids other than those indicated.

Location, Date, Signature