Analyzing Architectural Floor Plans Using Neural Networks

Balázs Szőke*
Supervised by: László Szécsi†

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

In this paper, we introduce a system that reconstructs a representation containing the positions and shapes of the walls, and the locations of the doors, from an input floor plan image (2D). We also build a 3D model from this obtained representation. To achieve this, we utilize traditional image processing algorithms to identify wall and door features, and convolutional neural networks based on differentiable rendering to reconstruct wall geometry. The purpose of our paper is to prove that the concept of differentiable rendering can help achieve a flexible and lightweight solution to this task.

Keywords: Deep Learning, Image Processing, Differentiable Rendering, Convolutional Neural Networks

1 Introduction

Architectural design workflow has shifted almost exclusively to computer-aided design software from traditional pencil and paper techniques. These modern tools are beneficial because it is cheaper and easier to handle digital media, and the design process is accelerated and more flexible. With the help of CAD systems, architects can utilize numerous tools to analyze plans, automate time consuming processes, e.g. generating 3D models from floor plans. To benefit from these possibilities, it is necessary to have a digital representation of the floor plan, but in some situations — e.g. handling archive documents — only a 2D image of the floor plan is available. Our goal with this paper is to introduce a system that is capable of overcoming these issues by not only creating the crucial digital representation from an input image, but also building a 3D model of the floor plan (Figure 1).
To achieve this, we use neural networks coupled with differentiable rendering. The flexibility of differentiable rendering makes this approach extensible to floor plans with more complex geometry, e.g. curved walls, or joints other than 90°.
In our paper, first, in Section 2, we give an overview of previous work in the field of analyzing architectural floor plans. In Section 3, we present our solution and describe the structure and idea behind the proposed method. Finally, in Section 4-5, we evaluate our method, and examine the weaknesses and potential development possibilities of our system.

2 Background

Converting floor plans to a digital representation is a well discussed problem in the field of computer vision. In the early 2000s, Dosch et al. [7] introduced a system that converts an input floor plan into a 3D model that contains the walls, doors, and furniture. To achieve this, they used traditional image processing algorithms. In our paper, we use a workflow similar to Dosch et al., but we extend it with neural networks. Macé et al. [13] focused on recognizing rooms. Ahmed et al. [1] improved this method further by separating and using text information from the floor plan to detect room function. Following the same semantic analysis approach, they enhanced their system by improving the text separation [3] and the room detection method [2].
With the rapid optimization of neural networks, more artificial intelligence-based approaches have shown up in this field. Dodge et al. [6] used convolutional neural networks and semantic segmentation [12] to recognize the walls. Liu et al. [11] gave a raster-to-vector method. Zeng et al. [15] improved the semantic analysis of floor plans with the help of neural networks, breaking the boundaries of some heuristic approaches that previous works used.
In the context of unsupervised learning, numerous different approaches have been examined that enhanced the effectiveness of training networks [5]. One of these methods is differentiable rendering [9], a very task-specific form of unsupervised learning, which yielded a solution to numerous complex computer vision problems. In our system we also utilize differentiable rendering in order to train a network that is capable of recognizing varied wall geometries.

3 Summary of our method

Our process can be divided into three main parts: 1) preprocessing the input image and separating the walls from

*szokeb@edu.bme.hu
†szecsi@iit.bme.hu

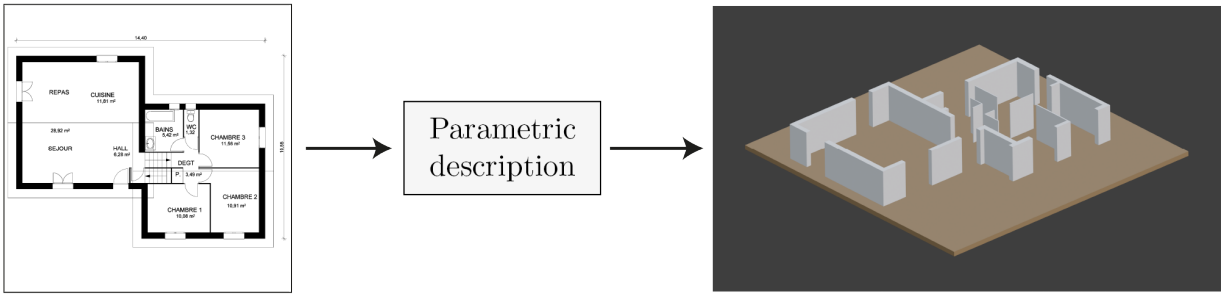

Figure 1: Overview of our system: from an input 2D image to a 3D model.

other symbols, 2) constructing a parametric representation of the walls with a neural network, 3) postprocessing of the parametric representation, and building the 3D model. In the first step, the goal is to reduce the input floor plan image to a cleaner, simpler binary image that contains only walls (no text labels or other architectural symbols on the image), and finding the position of the doors. After that, we split this image into smaller tiles with the same width and height. The reason behind this is to create a uniform input to the network regardless of the original dimensions of the floor plan, and more importantly, reduce the complexity of a single input, thus allowing us to simplify the requiered neural network architecture and the training process. The neural network processes all of these tiles and gives a parametric descriptor to a particular part of the floor plan (Figure 2). In the third part, we need to adjust and merge the individual parametric representations to eliminate the errors introduced by splitting walls running through multiple tiles in the original image. In this step we can insert doors at positions identified in the preprocessing phase. Finally, from the corrected parametric representation, we can build a 3D model of the floor plan.

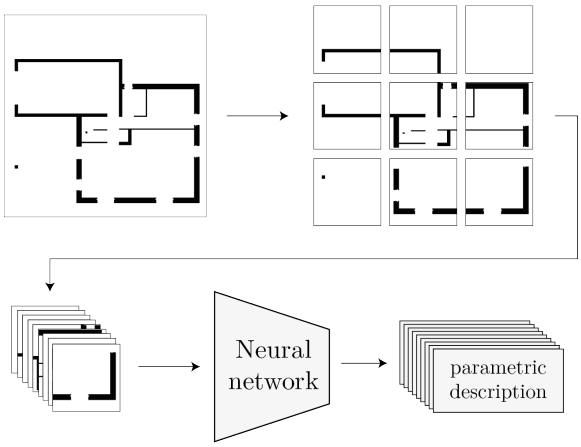

Figure 2: The high-level overview of the process of converting the input floor plan into a parametric descriptor.

3.1 Preprocessing

Following Ahmed et al. [2], we use mathematical morphological operators to separate different graphical symbols. Repeating erosion operators and then applying the same number of dilations eliminates thin lines from the image, removing text, furniture symbols, and other features other than walls. Splitting this into equally sized tiles, we have the input of the neural network.

To find the location of the doors, we targeted the circular arc of the door symbol. First, we need a view of the floor plan that shows only the thin elements of the original image, which we can get by subtracting the previously created "wall-layer" from the original image. To recognize arcs, we erased the straight lines, following Dosch et al. [7] and Ahmed et al. [2], using the Hough transform. After this, the image contains only curved lines, i.e. doors and parts of curved furniture (e.g bathtubs). To eliminate the remaining elements that are not parts of doors, we used a heuristic filtering condition based on the dimension difference between doors and other objects. For filtering, we used connected-component analysis.

3.2 Wall parametrization

Our goal in this step is to construct the parametric representation that describes the wall geometry within a floor plan tile. Analyzing generic, basic floor plans, we found that walls have uniform width and usually their endings are a rectangular cuts. Thus, we model wall segments as rectangles. Using this model, we can state a more accurate, low-level goal during the parametrization: approximating a complex shape in a binary image with rectangles, so that the union of the rectangles is as similar to the original shape as possible.

The neural network's input is a binary image of fixed dimensions and its output is the parameter set of the rectangles that approximate the input image. These parameters for each rectangle are: the endpoints of the wall segment (points in a coordinate system whose origin is the center of the image), and its width. For the implementation of the neural network, we used TensorFlow 2.0[1] and the Keras API[2], and for testing and evaluation, we used the CVC-FP database [4].

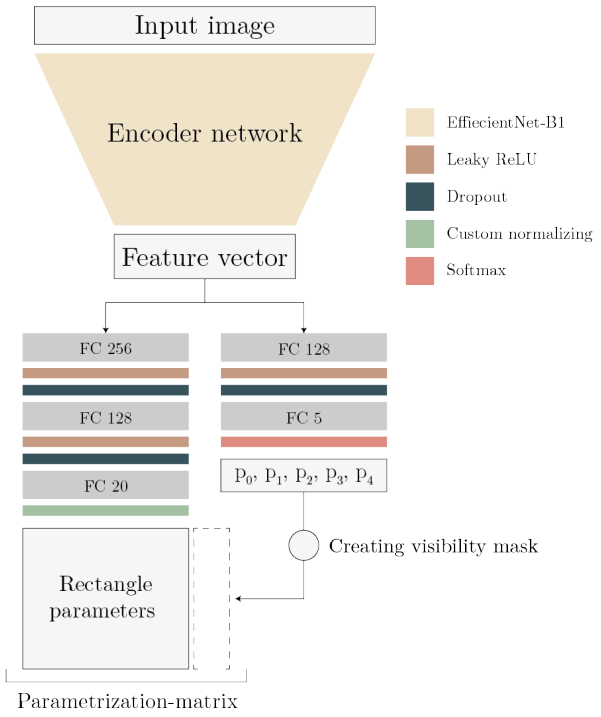

Figure 3: The architecture of our network.

3.3 Structure of our network

Our network can be divided into three main components: (Figure 3)

1) an encoder part which compresses the input image into a feature vector (N_{enc}),

2) a part which identifies the number of rectangles required to cover the original shape (N_{num}),

3) and a part which is responsible for finding the parameters of the rectangles that approximate the input (N_{param}).

As its input, the network accepts fixed 256×256 pixel wide, 1-channel images. First, the network compresses the input image into a feature vector which will be the input to two other parts of the network. The inside of the network branches in two directions. In our solution, the encoder part is the EfficientNet-B1 model [14].

In order to handle the varying number of the rectangles that is necessary to cover the input, the network's output is a parameter-matrix with dimensions of $N \times M$, where N is the maximum number of the rectangles, and $M = 6$ is the number of the parameters that describe a rectangle: the endpoint coordinates (2-2), the width of the wall, and a binary visibility value. The last parameter is required so that the output can be of fixed size, regardless of the number of rectangles required to cover the original shape. We chose $N = 4$ for the maximum number of rectangles. This is a heuristic limit; the reason behind this choice is that the tiling step in the preprocessing part (Section 3.1) produces inputs where only a very small part of the floor plan is present, thus the complexity of the visible region is strongly reduced, and 4 rectangles should be enough to represent a tile like this. In the output parameter-matrix, the rows are the parameters of the rectangles and the values of 1 in the last column show which rectangles are taking part in the process. This last column is the output of the subnetwork that identifies the required number of rectangles (N_{num}): an N-length vector which contains as many ones as rectangles needed, and the rest of the vector elements are zeros. To create this, the input feature vector goes through two fully-connected layers with a Dropout layer in between. The first layer has 128 neurons and Leaky-ReLU activation. The second layer has $N+1$ neurons ($0..N$ possible rectangles), and to create the probability distribution, a Softmax activation. To obtain the requiered output, we create an N-length mask from this probability vector based on the highest probability (*argmax*).

Another subnetwork (N_{param}) processes the feature vector into an $N \times (M-1)$ matrix, which represents the parameters of the rectangles that describe the input image. The output matrix will be extended with the visibility-mask, the output of the process described above. This part of the network contains three fully-connected layers; the first two layers have 256 and 128 neurons, and the last one has 4×5 neurons (maximum 4 rectangles, each with 5 parameters). Between the first two layers is a Dropout layer, and after each layer there is a Leaky-ReLU activation. To normalize the values of the last layer, we made a custom normalizing layer, which constrains the values to the appropriate range. The coordinates are normalized with a Tanh activation, guaranteeing that the endpoints are between the borders of the image (defining the origin as the center of the image), and the width with a Sigmoid activation, adjusting the output to the expected range: scaling with a maximum width value and translating with a minimum width value.

3.4 Training data

Creating an extensive training dataset of wall layout images from real-world architectural floor plans is an expensive and challenging task because of the limited availabil-

[1] https://www.tensorflow.org/
[2] https://keras.io/

ity and diversity of the data. However, because of the properties of the input — binary images that show wall segments (rectangles) —, it is possible to generate inputs in an algorithmic way. This is a serious advantage during the training process, since we can have full control over the properties of the input images, and we can create a theoretically infinitely large dataset.
To achieve this, we need an algorithm that can generate binary images that show various numbers of rectangles with random rotation, shape, and position. During the training process we used an algorithm with this functionality to create an endless source to the network, generating new inputs in every epoch.

3.5 Hybrid training method

When training neural networks, it is crucial to choose the right metrics to compare the output with the expected output. We need a well-constructed loss function that measures the error of the network output relative to a predefined expectation. Usually it is simpler, faster, and more efficient to use supervised learning than unsupervised methods. On the other hand, pre-defined expected outputs may be limited or inaccessible. To overcome these issues, we can utilize unsupervised learning, or reinforcement learning, where the loss function merely rewards outputs that conform to the input by some metric. This eliminates the need for defining explicit expected outputs during the training, thus making the training dataset size significantly smaller, and we have more freedom to provide training inputs to the network.
The generation of training data, described in Section 3.4, allows us to have an endless source of inputs, and also expected outputs (the parameters of the rectangles), since the generation process used these parameters to create images. But comparing these parameter matrices is not a trivial task. The issue is that we need to define an ordering of the rectangles (order of the rows in the parameter matrix), and although we can ensure a fixed order of the parameters in a single rectangle, there is no natural sorting for listing rectangles; in the context of the task, the order is irrelevant.
Furthermore, for a single rectangle, there are multiple combinations of parameters which code the same shape. For these reasons, the simple way, the trivial comparison between two matrices cannot work. By using complex, indirect measures, it is possible to compare the similarity of two rectangles, but the training process cannot rely entirely on this method, since minimizing this kind of loss would not guarantee perfect results.
To compensate this problem, we used unsupervised learning. Since the main goal of the network is to produce parameters that represent an image which is equivalent to the input image, the best metric for comparison is the visual similarity between the two images. In other words, we do not use the network's output in the loss function, but an image that is rendered from those parameters (Figure 4). To assure that the whole process is differentiable, the rendering step must also be differentiable. With this solution, we can eliminate the above mentioned issues, and train the network with more diverse geometries, and even with images of real floor plans — where it would require extensive expert labelling to define the correct expected parametric description. In our solution, we used a hybrid learning method, taking advantage of the benefits of both supervised and unsupervised learning.

3.6 Differentiable rendering

Implementing a differentiable renderer with the limited functionality we need for our goal is a fairly easy task. Our algorithm is a rectangle rasterizer, which fills a pixel matrix with values between 0 and 1 depending whether there is a wall in that pixel or not (more accurately, how likely it is for a wall to be there). It is important to use continuous, not discrete values, to have blur by the edges and thus gradients in the loss function. The amount of blur is controllable during the training, depending on the accuracy of the network. To have this blur effect by the edges of rectangles, we need a function that takes the coordinate of a point and returns the value of the corresponding pixel, in a way that the inside of the rectangles are constant 1 values, and the values of the area outside the rectangle are decreasing depending on the distance from the closest edge. Our method to calculate the blur is the following: Let the sides of the rectangles are on the $x = \frac{w}{2}$, $x = \frac{-w}{2}$, $y = \frac{h}{2}$, $y = \frac{-h}{2}$ lines, where w and h are the width and height of the rectangle (the width and length of the wall). The value of the pixel is:

$$\widehat{P_{ij}} = f\left(P_{ij}^x, \frac{w}{2}\right) * f\left(P_{ij}^y, \frac{h}{2}\right),$$

$$f(x,s) = \frac{1}{2} * \left(1 - \cos\left(e^{-\frac{\delta(x,s)}{\beta} - \log\frac{1}{\pi}}\right)\right),$$

$$\delta(x,s) = \max\{0, |x| - s\},$$

where $\widehat{P_{ij}}$ is the value of the pixel in the i-th row and j-th column of the image, $P_{ij} = \begin{pmatrix} P_{ij}^x & P_{ij}^y \end{pmatrix}$ is the center coordinate of the area that the pixel covers, β is the parameter that defines the amount of blur.
After rasterizing all N rectangles with the algorithm mentioned above, we have N images, one rectangle on each. We need to merge these into one final image. To achieve this, we used the following calculation to every pixel (P_{ij}^k is the pixel value on the k-th image):

$$P_{ij} = 1 - \prod_{k=1}^{N} 1 - \widehat{P_{ij}^k}$$

This formula guarantees that if an image already has a 1 in a certain pixel, then the same pixel in the final image will also be 1. Thus, during the merge, there is no rectangle that loses pixels, and every pixel value takes part in the final result in a differentiable way.

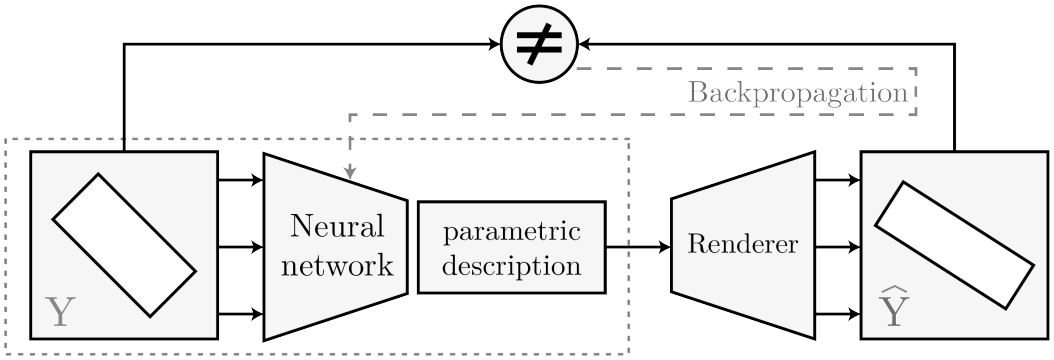

Figure 4: Overview of the learning process using a differentiable renderer. After the training, only the components inside the dotted area are used, thus we can deploy a relatively lightweight network.

3.7 Unsupervised loss function

During the training of the network, the dominant component of the loss function is the unsupervised metric. We create an image from the output of the network with the renderer described in Section 3.6, which we can naturally compare with the input image. The loss function for this is a mean squared error, which expresses the expectation of the most important goal of the whole process: the output should be as similar to the input as possible.

$$\mathscr{L}_{img} = \sum_{i=1}^{S}\sum_{j=1}^{S}\frac{(P_{ij}^{t} - P_{ij}^{p})^2}{S^2}$$

In the formula, S is the image size (1:1 ratio), P_{ij}^{t} and P_{ij}^{p} are the pixels in the i-th row and j-th column of the input and rendered image.

3.8 Supervised loss function

During the design process of the network architecture, it was a conscious choice to separate the part which identifies the necessary number of rectangles (N_{num}), in order to have a separate, more efficient loss function that targets this part. This part's output is a $N+1$ length probability distribution vector, which is easily comparable with the parameters that generated the corresponding input image. For the image generation, we used the same kind of parameter matrix descriptor as in Section 3.3, whose last column is a visibility mask. The expected number of rectangles is the sum of this vector. To measure the error of this part of the network we used a cross-entropy loss:

$$\mathscr{L}_{num} = -\sum_{i=0}^{N} Y_i^t * \log Y_i^p,$$

where Y^t is the expected number of rectangles in one-hot encoding, Y^p is the output of N_{num}, and the index i marks the i-th element of these vectors.

During the input image generation, the algorithm produces a possible descriptor of the image, which can be used to create a loss function that compares directly the output of the network and this parameter set. To avoid the wrong approach mentioned in Section 3.5, we compared rectangle parameters by similarity. For each rectangle descriptor, we find the most similar rectangle in the other parameter set. In other words, we need to find a minimum cost assignment between the two sets, where the edge weights are the similarity between rectangles. The sum of the edge weights that are included in the assignment is the loss: $\mathscr{L}_{param}$. To solve this problem, we used the matrix formulation of the Hungarian method [10], where the elements of the input matrix are:

$$M_{ij} = V_i^t * \psi(R_i^t, R_j^p) + \lambda * \phi(V_i^t, V_j^p),$$

where R_i^t and R_i^p are the i-th parameter-vectors of the target- and predicted output, V_i^t and V_i^p are the last elements of these vectors, $\psi()$, $\phi()$ are distance functions based on the properties and visibility of the rectangles.

The visibility value makes it harder to compare two rectangles, because a non-visible rectangle can have arbitrary position and orientation, which should have no effect. Thus, the obvious way, to measure the similarity by comparing the parameters element-wise, would not work. Comparing a visible and a non-visible rectangle, the main component of the distance should be the visibility difference, since the goal is the most similar visual result. In the formula above, the first constraint is realized in the possibility to cancel out the parameter distance function $\psi()$ with the V_i^t factor. If the i-th rectangle is non-visible, this factor is 0; otherwise it is a harmless 1. The second constraint is realized in the λ constant, amplifying the effect of the visibility distance function $\phi()$. This function is squared distance:

$$\phi(V_i, V_j) = (V_i - V_j)^2$$

To compare the shapes and positions of two rectangles, we need to use representation-invariant properties, to avoid the misleading behaviour of the raw parameters, e.g. endpoint coordinates drastically different if one of two

identical rectangles is mirrored, although visually there is no difference. These representation-invariant properties are: the area of the rectangle, the ratio of its sides, the coordinate of its center, the horizontal/vertical extension of the rectangle, and a heuristic metric about its rotation. Using these properties, the parameter distance function is the following:

$$\psi(R_i,R_j) = \Delta_a(R_i,R_j) + \Delta_{sr}(R_i,R_j) + \Delta_c(R_i,R_j) + \Delta_h(R_i,R_j) + \Delta_v(R_i,R_j) + \Delta_r(R_i,R_j),$$

$$\Delta_a(R_i,R_j) = \Delta(w_i * l_i, w_j * l_j),$$

$$\Delta_{sr}(R_i,R_j) = \Delta\left(\frac{\min\{w_i,l_i\}}{\max\{w_i,l_i\}}, \frac{\min\{w_j,l_j\}}{\max\{w_j,l_j\}}\right),$$

$$\Delta_c(R_i,R_j) = \Delta(c_i^x, c_j^x) + \Delta(c_i^y, c_j^y),$$

$$\Delta_h(R_i,R_j) = \Delta(bb_i^w, bb_j^w), \quad \Delta_v(R_i,R_j) = \Delta(bb_i^h, bb_j^h),$$

$$\Delta_r(R_i,R_j) = \Delta\left(\frac{w_i * l_i}{bb_i^w, bb_i^h}, \frac{w_j * l_j}{bb_j^w, bb_j^h}\right),$$

$$\Delta(x,y) = (x-y)^2,$$

where w, l is the width and length (height) of the rectangle, $\begin{pmatrix} c^x & c^y \end{pmatrix}$ is the coordinate vector of its center, bb^w and bb^h is the width and height of its bounding box. The individual functions, $\Delta_a, \Delta_{sr}, \Delta_c, \Delta_h, \Delta_r$ are the distance functions of the above mentioned representation-invariant properties respectively.

3.9 Hybrid loss function

Using the metrics described in Section 3.7-3.8, we utilized both supervised and unsupervised methods. The combined, hybrid loss function we used for training the network is the weighted sum of the three loss components, where the weights are controllable during the training process to achieve optimal behaviour:

$$\mathscr{L} = \alpha * \mathscr{L}_{img} + \beta * \mathscr{L}_{param} + \gamma * \mathscr{L}_{num}$$

3.10 Training heuristics

During the training process, we used widely applied automatic methods and intuitive, heuristic techniques which require more manual intervention, to control the training. To manage the network weights, we chose the Adam optimizer algorithm, and to control the learning rate, we used a discrete staircase decay. The amount of blur applied during the rendering process is relatively high at the start of the training, to assure overlapping between rectangles, and slowly decreased as the accuracy of the network is increased, to achieve a more accurate result. This decreasing was done manually, analyzing the progress of improvement. The most well-known problem when it comes to training neural networks is the issue of overfitting. Luckily, since we can use the input generating algorithm presented in Section 3.4, we can completely avoid any form of overfitting.

3.11 Postprocessing of the parameters

Due to splitting the original floor plan into smaller parts before processing with our network, a wall can appear in multiple tiles, which is completely unrecognizable to the network, since it sees only one floor plan tile at a time. Thus, the network covers these walls with multiple smaller wall-segments instead of a more accurate, longer segment. These smaller segments can have different width and horizontal/vertical axis, which can be adjusted after the parametrization step. The goal in this step is to overcome this problem and to merge the individual parametrizations into one that is more accurate in the context of the whole floor plan. At this step, we can integrate the door positions into the parametric representation. Because of the imperfections in the network's output, we need to adjust the door coordinates, and snap it to the nearest wall opening. If we can define a minimum or maximum door width value, then we can adjust the wall endings, too, according to this value. After these steps, we can build a 3D model from the parametrization of the walls, using a custom Python script, which creates a 3D scene in Blender[3].

4 Results

The presented system is capable of creating a parametric representation of the floor plan and a 3D model, using only an input image, and we showed that the concept of differentiable rendering can be used to achieve this. During the evaluation of the results of our system, we used mostly qualitative evaluation, since the main goal was to have a visually similar result to the input image.

In the first step, when we separate the walls, we fine-tuned the number of repetitions of the morphological operators to have the most optimal results on the database we used, and this method barely makes any errors; the separation is clean in most cases. During the finding of the door positions, the other symbols that intersect with the door symbols have the biggest impact on the quality of our heuristic approach. On a generic, non-overcrowded floor plan, we recognized doors with high accuracy by tuning the parameters of the algorithm in an empirical way, but it made noticeable errors when the arch of the door was intersected by other lines, or only a tiny part of it was visible.

The wall parametrizer neural network worked with very

[3] https://www.blender.org/

high accuracy in those cases when the input formation was simple, as expected, and its accuracy decreases as the complexity of the shape on the input image increases. But in general, if the complexity is equal to a part of a real floor plan, the results are good enough to be a visually similar representation of the original, after the postprocessing step (Figure 5). The network is very accurate at capturing the shape of the walls, the noticeable errors occurring mostly because of the imperfect rotation angle. The network makes more mistakes in cases where a wall endpoint is close to the edge of the image, mainly because of the imperfect training data generating algorithm.
The part of the network, which identifies the required number of rectangles, works with high accuracy, but we can see a decreasing tendency here too, as the complexity of the input shape increases. The training data generating algorithm has a big impact on this, because the imperfect generation sometimes results images, where some rectangles are visually not separable from the other rectangles, or other rectangles cover them completely. In these situations, the network, in most cases, correctly identifies the number of visibly separable rectangles, but the label of the data marks a different number; thus the loss is not always correct. After manually correcting these labels during the evaluation, the confusion matrix of this part is the following:

	0	1	2	3	4	Accuracy
0	22%	0%	0%	0%	0%	100%
1	0.4%	17.6%	0%	0%	0%	98%
2	0%	1%	18%	0%	0%	95%
3	0%	0.1%	1.4%	16%	1%	86%
4	0%	0%	1%	3%	18.5%	82%

Table 1: The confusion matrix of the network. On the horizontal axis is prediction of the network; on the vertical axis is real number of rectangles.

By optimizing the parameters of the postprocessing step, significant improvement can be achieved in those cases where the neural network works with a lower accuracy. It can efficiently compensate the errors of the wall rotations, and the difference in wall widths, thus reducing the necessary level of accuracy of the network which speeds up the training process.

For training the network, we used the Google Colab[4] cloud based service, which makes hard to provide exact specification on the hardware used during the process, but it took about 12 hours to achieve the above presented results.

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

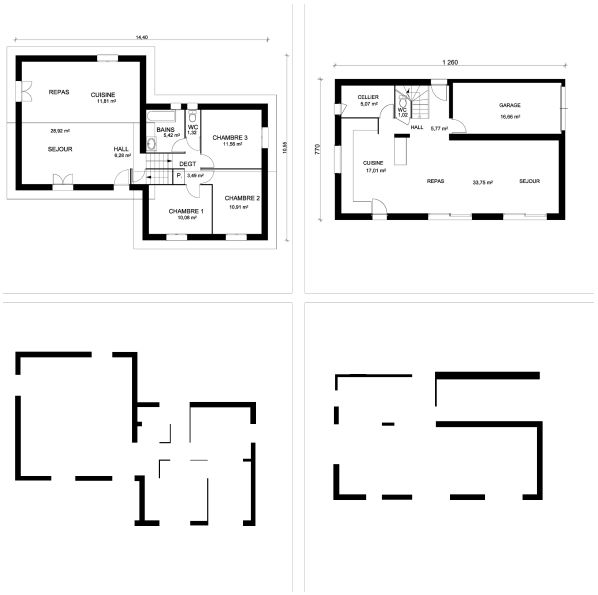

Figure 5: The results of our network. In the top row are the original floor plans from the CVC-FP database, and in the bottom row are images rendered from the parametric representation that the network produced.

5 Future work

There are several points where the system can be improved to have a better result. The most important part of the system is the neural network, this is the point where there is the most room for improvement. The network layers and parts were not fully optimized, and for optimal results, it would be beneficial to find the best parameters. It would be beneficial to integrate the concept of splitting the input image into the network, e.g. with higher level convolution or with the use of RNN models [8]. Using reinforcement learing techniques to improve the training efficiency would be a logical extension of the current approach. Perfecting the training data generating algorithm is crucial to eliminate the problems in the part that identifies the required number of rectangles. Finding faster, more efficient algorithms in the preprocessing step can improve the speed of the system, since 90% of the process time is spent on this step. The system can be extended with algorithms to recognize more symbols in the preprocessing step, to reserve more information from the original floor plan, e.g. furniture.

6 Acknowledgements

This work has been supported by OTKA K-124124. The research presented in this paper, carried out by BME, was supported by the Ministry of Innovation, and the National Research, Development and Innovation Office, within the framework of the Artificial Intelligence National Labora-

tory Programme.

References

[1] Sheraz Ahmed, Marcus Liwicki, Markus Weber, and Andreas Dengel. Improved automatic analysis of architectural floor plans. In *2011 International Conference on Document Analysis and Recognition*, pages 864–869. IEEE, 2011.

[2] Sheraz Ahmed, Marcus Liwicki, Markus Weber, and Andreas Dengel. Automatic room detection and room labeling from architectural floor plans. In *2012 10th IAPR international workshop on document analysis systems*, pages 339–343. IEEE, 2012.

[3] Sheraz Ahmed, Markus Weber, Marcus Liwicki, and Andreas Dengel. Text/graphics segmentation in architectural floor plans. In *2011 International Conference on Document Analysis and Recognition*, pages 734–738. IEEE, 2011.

[4] Lluís-Pere de las Heras, Oriol Ramos Terrades, Sergi Robles, and Gemma Sánchez. CVC-FP and SGT: a new database for structural floor plan analysis and its groundtruthing tool. *International Journal on Document Analysis and Recognition (IJDAR)*, 18(1):15–30, 2015.

[5] Happiness Ugochi Dike, Yimin Zhou, Kranthi Kumar Deveerasetty, and Qingtian Wu. Unsupervised learning based on artificial neural network: A review. In *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, pages 322–327. IEEE, 2018.

[6] Samuel Dodge, Jiu Xu, and Björn Stenger. Parsing floor plan images. In *2017 Fifteenth IAPR international conference on machine vision applications (MVA)*, pages 358–361. IEEE, 2017.

[7] Philippe Dosch, Karl Tombre, Christian Ah-Soon, and Gérald Masini. A complete system for the analysis of architectural drawings. *International Journal on Document Analysis and Recognition*, 3(2):102–116, 2000.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020.

[10] Harold W Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[11] Chen Liu, Jiajun Wu, Pushmeet Kohli, and Yasutaka Furukawa. Raster-to-vector: Revisiting floorplan transformation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2195–2203, 2017.

[12] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[13] Sébastien Macé, Hervé Locteau, Ernest Valveny, and Salvatore Tabbone. A system to detect rooms in architectural floor plan images. In *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, pages 167–174, 2010.

[14] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[15] Zhiliang Zeng, Xianzhi Li, Ying Kin Yu, and Chi-Wing Fu. Deep floor plan recognition using a multi-task network with room-boundary-guided attention. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9096–9104, 2019.