Software Engineering Department

Braude College of Engineering

Capstone Project Phase B – 61999

Relational Generative Adversarial Networks for Graph-Constrained House Layout Generation

23-2-R-7

Yael Shusterman 205804446

Yael.Shusterman@e.braude.ac.il

Yeela Malka 207727744

yeela.malka@e.braude.ac.il

Supervisors:

Ph.D. Dvora Toledano – KitaiProf. Zeev Volkovich

Abstract

A floor plan is a scaled diagram of a room or building viewed from above.

The floor plan must be planned in accordance with the client's restrictions as a crucial component of architectural design. Typically, the client defines the intended layout of the home in terms of the different room types and how they relate to one another in space, and the architect creates the plan in response. To reduce expenses and avoid compromises between the customer and the architect about the house's design, there have been numerous attempts to use artificial intelligence to speed up this long and iterative procedure.

This project introduces the House-GAN, whose goal is to take the client's preferences as graphical input and produce a collection of axis-aligned bounding boxes for the rooms, which the architect will then convert to a floor plan.

Index Terms: Deep learning; Generative Adversarial Network; Graph-constrained; layout; Generation; Floorplan

1.Introduction

1.1 Motivation for an automatic design process of a house floor plan

There are several steps involved in designing floorplans for a house. Firstly, the customer consults an architect to create floor plans. The architect considers the types of rooms, the number of rooms, and of course, which rooms should be adjacent to each other. In a typical procedure, architects create ‘bubble diagrams’ using the above considerations. Then, the floor plans are generated and modified multiple times based on customer feedback. This entire process is time consuming and expensive. Thus, a demand for automating this process is created.

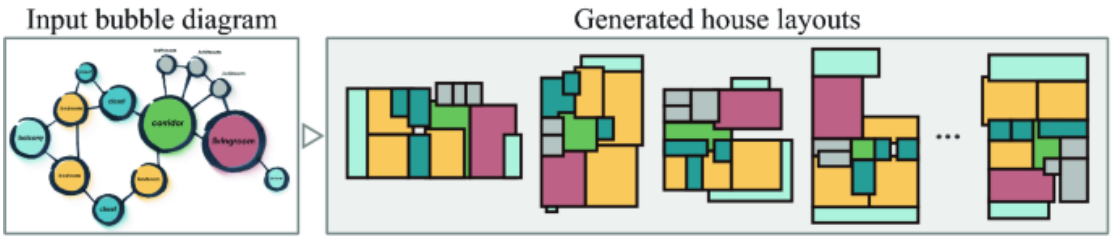
An example Bubble Diagram

1.2 How can GANs be used to automate this process?

Generative Adversarial Networks or GANs are popular in the image processing field for producing realistic images of humans, handwriting, etc. GANs have also proven to be effective for image generation with constraints. In this case, the bubble diagram is modeled as a graph with constraints. The House-GAN model proposed in the paper employs a relational generator and discriminator with the graph structure containing the encoded constraints. By using convolutional message-passing neural networks (Conv-MPN), the adjacency constraints of the rooms can be effectively captured.

1.3 The problem statement

The problem can be formulated as a development of a GAN that takes an architectural constraint as a graph as input and produces a set of axis-aligned bounding boxes of rooms. In this graph, each room is represented as a node and the type of room is embedded as node information. The spatial adjacency is represented as an edge between two nodes. The quality of output, i.e., generated house layouts is measured using the three metrics: realism, diversity, and compatibility with the input graph constraint. The qualitative and quantitative evaluations are performed over 117,000 real floorplan images from the LIFULL Home dataset.



2. Related Work

Layout composition has been a topic of active research in areas such as architectural layouts [1,2,3,4], game-level design [5,6], and others. Peng et al. [4] and Ma et al. [6] are examples of more traditional methods for generating diverse layouts. House-GAN uses stronger data-driven techniques.

Wuang et al. [7], Ritchie et al. [8], Wu et al. [9], Jyothi et al. [10], and Li et al. [11] propose data-driven methods that use GAN for automatic floorplan generation, which yield effective methods. But unlike the House-GAN model, they are unable to accept a graph as an input.

Research has also concentrated on layout generation under graph constraints. From input graphs, Wang et al. [12], Merrel et al. [13], Jonhson et al. [14], and Ashual et al. [15] produce layouts and create realistic graphics.

The relational GAN used in the creation of the House-GAN model encrypts the input constraint into the graph structure of the relational generator and discriminator. The qualitative and quantitative evaluations show the House-GAN approach's efficacy.

3. Background

Theoretical and mathematical foundations for the model's implementation are discussed in this section.

3.1 Neural Network

A Neural Network (NN) is a type of machine-learning algorithm that seeks to mimic the structure and function of the human brain. It is made up of interconnected layers of "neurons" that process and send data. The weights of the connections between neurons are changed during the training process.

NN is especially beneficial for jobs that are difficult to complete with standard systems since they can learn and adapt to patterns in data on their own.

To train a Neural Network, a large dataset of instances to learn from is required, as is a procedure for altering the weights of the connections between neurons based on the network's output.

3.1.1 Backpropagation

Backpropagation, also known as backward propagation of mistakes, is a test for errors that works backward from output nodes to input nodes. It is a useful mathematical tool for boosting prediction accuracy in data mining and machine learning. Neural Networks use backpropagation to compute a gradient descent regarding weight values for diverse inputs. The systems are tuned by modifying connection weights to narrow the disparity between desired and achieved system outputs as much as possible.

3.1.2 Loss function

The loss function computes the difference between the NN algorithm's current output and the expected output. It used for assessing how well the algorithm models the data.

3.1.2.1 WGAN loss function

A loss function based on the Wasserstein distance concept, which estimates the distance between probability distributions. It quantifies the difference between the real and produced data distributions in the context of GANs.

3.1.2.1.2 Wasserstein distance

Wasserstein distance is a mathematical measure of the dissimilarity between two probability distributions. It is also known as Earth-Mover (EM) distance or Kantorovich-Rubinstein norm. It quantifies the bare minimum of "work" or "effort" required to convert one distribution to another.

$$D_W(P \| Q) = \underbrace{\inf_{\gamma \in \prod(P,Q)} E_{(x,y)\sim\gamma}}_{\text{"Transport plan"}} \underbrace{\left[\|x-y\|\right]}_{\text{Euclidean Distance}}$$

Lower bound

Wasserstein Dinstance

Fig.1: Wasserstein distance calculation.

3.1.3 Relational Neural Network

A Relation Network (RN) is a component of an artificial neural network with a structure that can reason about relationships between objects. Spatial relations (above, below, left, right, in front of, behind) are one type of such relation.

RNs can infer relationships, are data efficient, and can operate on a set of items regardless of their order.

3.2 GAN

Generative Adversarial Networks are generative models, that generate new data instances that are resemble the training data.

GANs accomplish this level of realism by combining a learning generator to produce the goal output with a learning discriminator to differentiate true data from the generator's output. The generator attempts to mislead the discriminator, while the discriminator attempts to avoid being tricked.

Neural Networks are used in the generator and the discriminator as well.

The generator improves its ability to generate plausible data. The instances that are generated serve as negative training examples for the discriminator.

The discriminator learns to discriminate between actual and bogus data generated by the generator. The generator is penalized by the discriminator for creating improbable results.

When training begins, the generator generates blatantly bogus data, and the discriminator soon learns to recognize it.

Finally, if generator training is successful, the discriminator becomes worse at distinguishing between real and false. It begins to misclassify bogus data as real, and its accuracy decreases as a result.

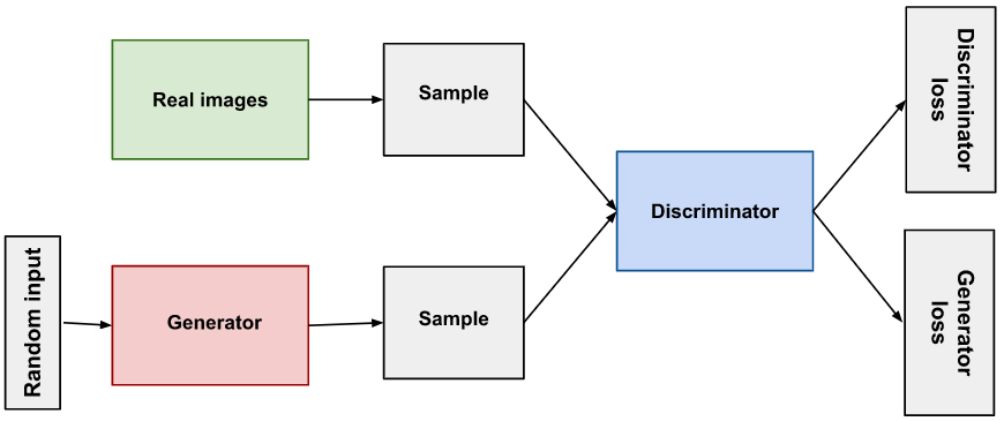


Fig.2: GAN structure

3.2.1 Discriminator

The training data for the discriminator originates from two sources:

- Real data instances, such as real pictures of people. The discriminator uses these instances as positive examples during the training phase.

- Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

Discriminator training:

- Two loss functions are connected to the discriminator. The discriminator ignores the generator loss and only uses the discriminator loss during training. The generator loss is used during generator training.

- The discriminator distinguishes between actual and fake data generated by the generator.

- The discriminator loss penalizes the discriminator for incorrectly categorizing a real instance as a fake instance or a fake instance as a real instance.

- Backpropagation from the discriminator loss through the discriminator network changes the discriminator's weights.

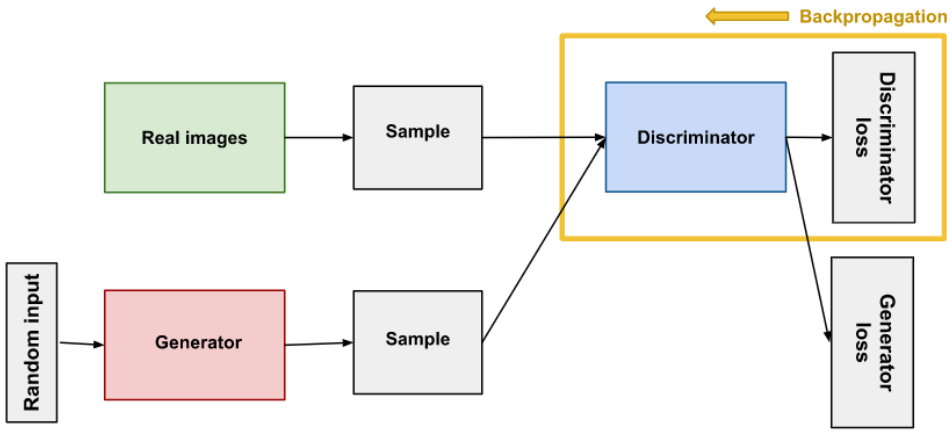


Fig.3: Backpropagation in discriminator training.

3.2.2 Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input

- generator network, which transforms the random input into a data instance

- discriminator network, which classifies the generated data

- discriminator output

- generator loss, which penalizes the generator for falling to fool the discriminator

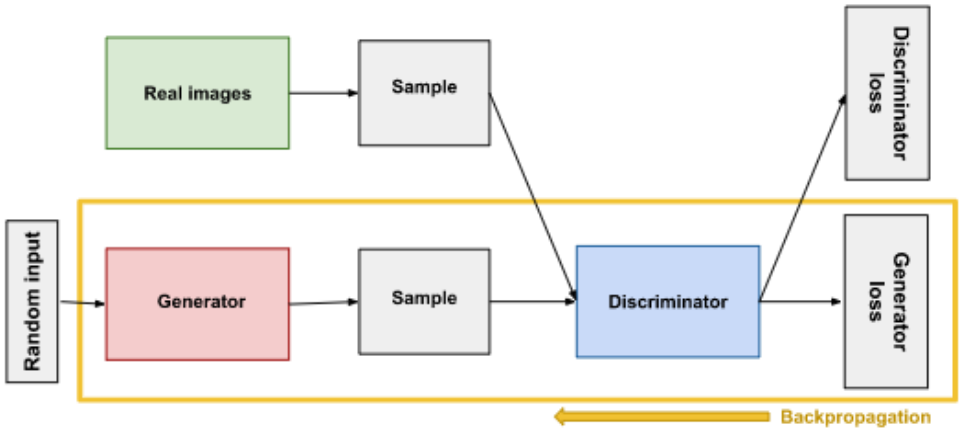


Fig.4: Backpropagation in generator training.

3.3 CNN – convolutional neural network

Convolutional Neural Networks (CNNs) are neural networks that are specifically intended for processing data having a grid-like layout, such as photographs. These networks are particularly useful for image identification and processing jobs because they can learn visual features and patterns by examining pixel correlations, a process called as feature extraction.

CNNs are very useful for picture classification, object detection, and segmentation because they can learn hierarchical features by applying many filters to the input data. These filters enable the network to recognize patterns and features at different levels, from low-level features like edges and corners to higher-level features like shapes and objects. CNN have three main types of layers, which are:

- Convolutional layer

- Pooling layer

- Fully connected layer

3.3.1 Convolutional layer

The convolutional layer is the foundation of a CNN and is where most of the processing takes place. It necessitates a few components, including input data, a filter, and a feature map. It also has a feature detector, sometimes known as a kernel or a filter, which traverses through the image's receptive fields, checking for the presence of the feature. This process is known as a convolution.

The feature detector is a two-dimensional weighted array that represents a portion of the image. The filter size is commonly a 3x3 matrix, which also defines the receptive field size. The filter is then applied to a portion of the image, and the dot product between the input pixels and the filter is computed. This dot product is then supplied into an array of outputs. The filter then shifts by a stride and repeats the operation until the kernel has swept across the entire image. A feature map, activation map, or convolved feature is the result of a series of dot products from the input and the filter.

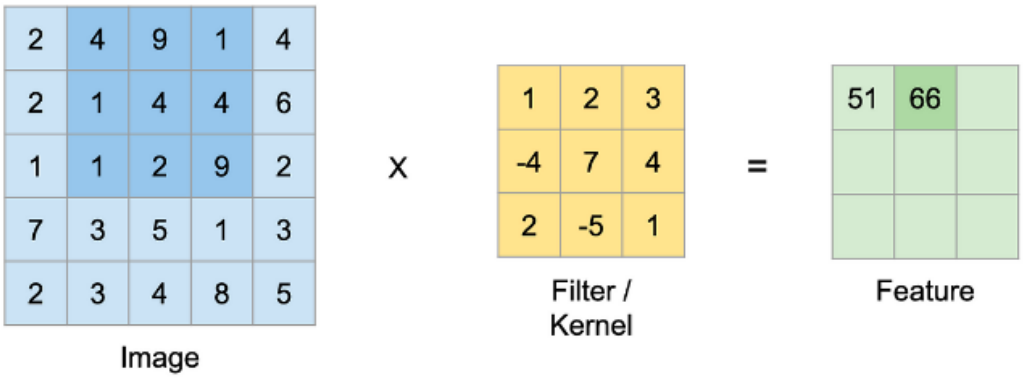


Fig.5: Convolution Operation.

Following each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

3.3.1.2 ReLU

The rectified linear activation function, abbreviated ReLU, is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. Any negative values in the filtered image are replaced with zeros in this layer. When the node input surpasses a certain threshold, this function is activated. As a result, the output is zero when the input is less than zero. When the input exceeds a certain threshold, it has a linear relationship with the dependent variable. This means that, in order to avoid summing with zero, it can enhance the speed of a training data set in a deep neural network faster than other activation functions.

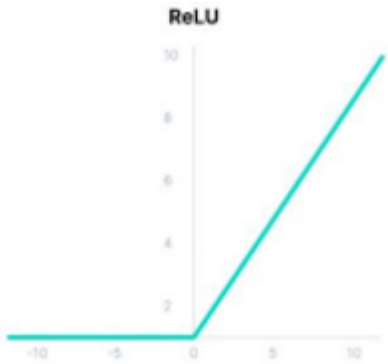


Fig.6: ReLU function.

3.3.1.3 Leaky ReLU

Because all negative gradient values become 0, the weights for some neurons are not updated during backpropagation, this is called the dying ReLU problem.

When there is a lot of noise or outliers in the data: Leaky ReLU can yield a non-zero output for negative input values, which might help to prevent rejecting potentially significant information (as mentioned above in the dying problem).

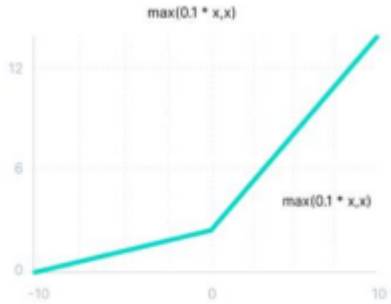


Fig.7: Leaky ReLU function.

3.3.1.4 Activation function

An Activation Function determines whether a neuron should be activated or not. This means that it uses simpler mathematical operations to determine whether the neuron's input to the network is essential or not throughout the process of prediction.

3.3.2 Pooling layer (Downsampling)

Pooling layers, also known as Downsampling, reduces the number of parameters in the input by performing dimensionality reduction. The pooling process, like the convolutional layer, sweeps a filter across the entire input, but this filter does not have any weights. Instead, the kernel applies an aggregation function on the receptive field values, creating the output array. While much information is lost in the pooling layer, it does have several advantages for the CNN. They aid in the reduction of complexity, the enhancement of efficiency, and the reduction of the risk of overfitting.

3.3.2.1 Max Pooling

Type of pooling: as the filter moves across the input, it selects the pixel with the maximum value to send to the output array.

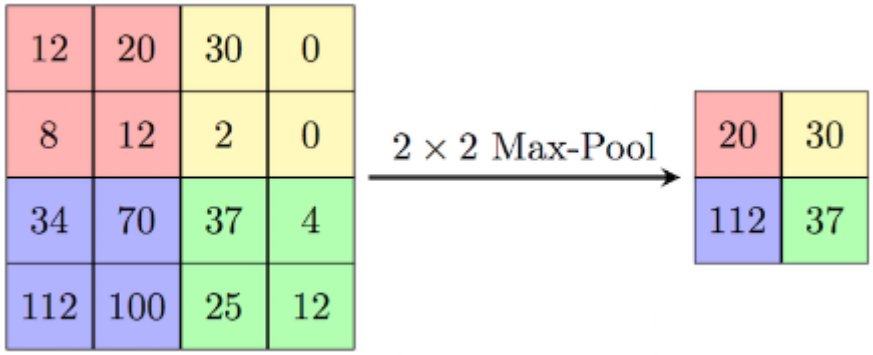


Fig.8: Max Pooling Operation.

3.3.2.2 Sum Pooling

Type of pooling: as the filter moves across the input, it calculates the sum of all the pixels in the window, and each element in the resulting pooled feature map represents the sum of the values in the pooling window in the input feature map.

3.3.3 Fully Connected layer

Each node in the output layer connects directly to a node in the preceding layer in the fully connected layer.

This layer performs classification based on the features retrieved by the previous layers and their various filters.

3.4 Message Passing Networks

Convolutional Neural Networks (CNNs) have proved to be a great way to deal with visual data. However, in many real-world problems, the input data for a problem is not easily captured in such a highly regular format as an image. Information can be more effectively captured if the data is represented in a graph form instead. For example, connectivity of the rooms can be modelled as graphs. Wouldn't it be great if there was a way to create deep learning models that can take a graph of arbitrary size as input? This is where Message Passing Networks (MPNs) come in.

The idea of MPNs is simple. Each node has a hidden state in the form of a vector. Then, the hidden state of each node is updated simultaneously, by using 'messages' constructed based on the hidden states of each adjacent node. Then, this process is repeated for a fixed number of rounds. In the end, the output of an MPN is the same graph as the input graph, only with updated hidden states that might incorporate information about the structure of the graph, the relation of each node to its neighbors, or its neighbors' neighbors, etc. That depends on how you the messages and the loss function are designed.

The model is aimed to be invariant to the input graph size and number of edges, so that it can be presented by any graph structure. Therefore, in the case of MPN, the same function and (learned) parameters are generally used for each of the node updates. "node updates" are performed as follows.

To update a single pixel for the next layer in CNN, a kernel a kernel could be reused across the entire image. This use of a kernel over data with high regularity such as an image can be thought of as a very specific case of convolutions in a graph, as shown on the left in **Figure 9**. Now, the problem for 'convolutions' arbitrary graphs can be easily seen as as shown on the right in **Figure 9**. I.e., how to define a 'kernel' that deals with 1) a variable number of neighbors and 2) no implicit ordering of the neighbors (in an image you know which neighbor is top-left, top-center, bottom-right, etc.).

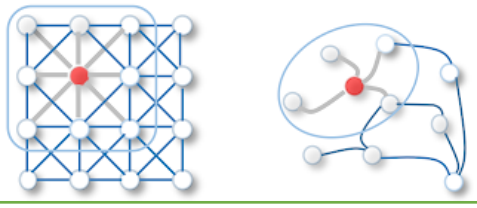


Fig.9: Convolutions (left) can be thought of as a special case of graph convolutions (right).

This all depends on how the messages are combined. For the hidden state h_i^l of a particular node i at round l, a message ψ is constructed for each of its neighbors. This function ψ is a design choice and has learnable parameters. The new state of the node at round $(l+1)$ is based on its own current stated and a combination of all the messages. Once again, the function φ is a design choice and its parameters can be learnable. The important part here, is

that the message combinator $\oplus$ is permutation invariant, e.g., summation or max pooling. This ensures that the order in which the nodes are stored does not affect the output of the MPN. In one formula, a single node update looks like this (where $N(i)$ are the neighbors of i):

$$h_i^{(l+1)} = \phi\left(h_i^{(l)}, \bigoplus_{j \in N(i)} \psi\left(h_i^{(l)}, h_j^{(l)}\right)\right)$$

3.5 Conv-MPN

A variation of MPN named Conv-MPN [16] are used in HouseGAN model. The key difference compared to regular MPN is that Conv-MPN uses feature volumes instead of feature vectors as node states. To construct messages, it uses a CNN architecture. Conv-MPN are used in HouseGAN, since the task they try to solve is spatial in nature, thus its make sense to use of a hidden state with at least 2 dimensions.

3.6 Floorplan Vectorization

Floorplan Vectorization deals with Converting a rasterized floorplan image into a vector-graphics representation.

Since the database used to train the model in the House-GAN model does not contain bubble diagrams, the floorplan vectorization algorithm is used to generate the vector-graphics format, which is then turned into a bubble diagram (the model input).

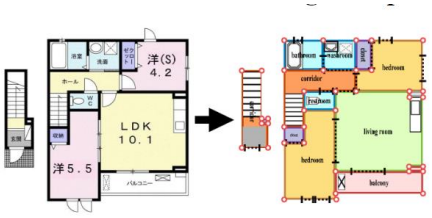


Fig.10: from raster to vector.

3.6 Manhattan distance

The Manhattan distance is the distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $|x_1 - x_2| + |y_1 - y_2|$(norm 1 distance).

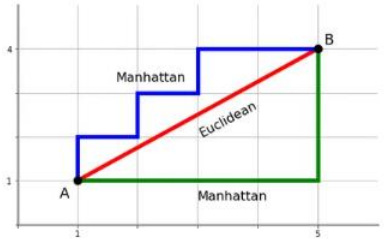


Fig.11: Manhattan distance.

3.7 K-fold validation

When given new data, K-fold Cross-Validation divides the dataset into K folds and is used to test the model's ability. K denotes the number of groups into which the data sample is divided. For example, if the K-value is 5, it is referred as a 5-fold cross-validation.

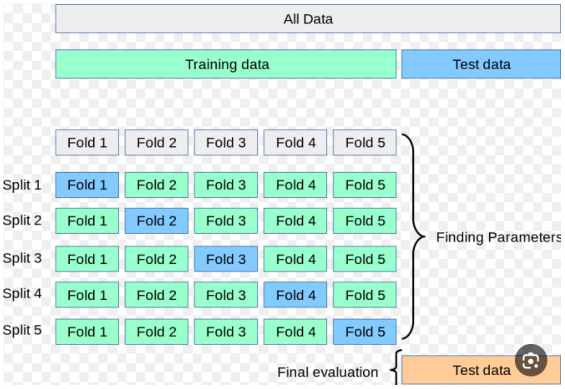


Fig.12: K- fold Cross Validation

3.8 Upsampling Layer

An upsampling layer is a component in Neural Network topologies that is widely used to increase the spatial dimensions or resolution of feature maps. It is the inverse of downsampling or pooling layers, both of which lower spatial dimensions. When it is required to recover or restore the original spatial information lost during downsampling, or to raise the resolution of the feature maps for better discrimination and finer detail representation, upsampling layers are used.

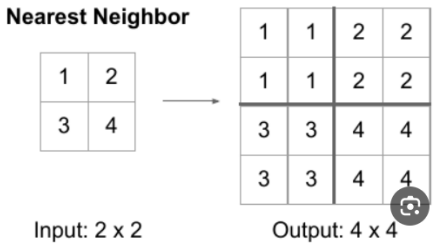


Fig.13: Nearest Neighbor Upsampling

3.9 Noise Vector

A noise vector is a randomly produced input vector that acts as a source of stochasticity or unpredictability for generative models such as generative adversarial networks (GANs). It is required for the generative model to generate different and realistic samples.

A noise vector is often a high-dimensional vector drawn from a probability distribution, most commonly a standard normal distribution (mean 0, variance 1). The vector's elements each reflect a separate source of randomness or variation. It is feasible to generate a wide range of distinct outputs by feeding different noise vectors into the generative model.

4. The Model

4.1 The Model Overview

The full model consists of a series of preprocessing steps followed by a Generative Adversarial Network (GAN). A schematic overview of the model is represented in Fig 14. The data preprocessing steps, and the actual network are discussed in separate sections.

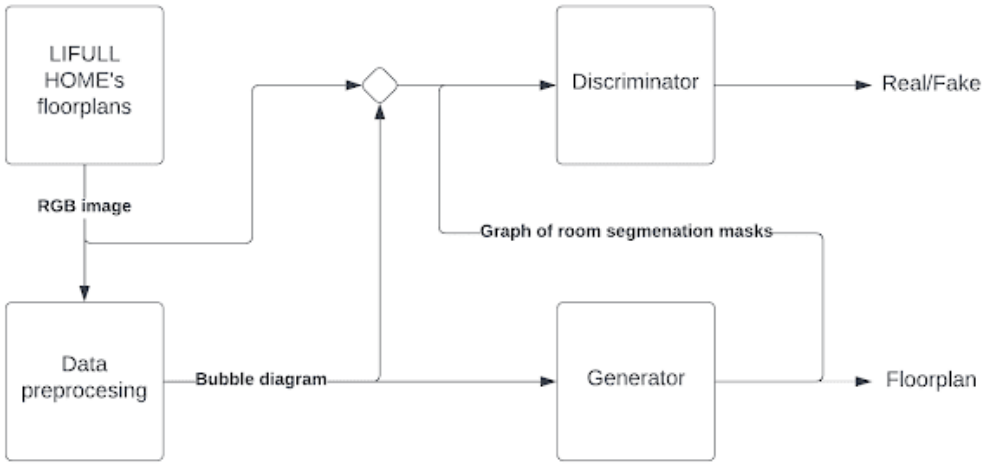


Fig.14: Model Overview

As can be seen from the diagram the generator's output is used in two different scenarios. It is used to construct an actual floorplan during inference and use it as input data for the discriminator during training. Given these two usages, one might ask the following questions:

1. How the generator's output can be converted to a realistic floorplan during inference?
2. How raw (real) floorplan RGB images can be firmed to the discriminator's input format during training?

4.1.1 Floorplan constructing from generator output

In short, rectangle fitting is performed on room segmentation masks generated for each room. This is elucidated in the Data post-processing section.

4.1.2 Discriminator input construction from real floorplan

A graph of room segmentation masks must be constructed from a real floorplan to provide real data to the discriminator. The data preprocessing step makes sure the RGB floor plan image is converted to a bubble diagram. The segmentation mask is created from the RGB image by setting the foreground (i.e., the room) to 1.0 and the background to -1.0. This is done for each node in the graph, the result is a graph of room segmentation masks which can be fed to the discriminator.

4.2 Data preprocessing

Constructing a model to generate house layout requires data to train on. The research uses the LIFULL HOME's database [17]. This dataset contains five million real RGB floorplan images of which the paper selected 117,587. A few preprocessing steps are applied to these raw RGB images before they are being used by the model.

The first step is to uniformly rescale all floorplan images to fit inside the 256x256 resolution. After this rescaling, a vectorized version of the floorplan is created. This is done by a floorplan

vectorization algorithm. The last step is to convert this vectorized floorplan in a bubble diagram. The bubble diagram is used as input for the model.

In this preprocessing pipeline two important steps are done which require more explanation.

4.2.1 Constructing a vectorized floorplan

Constructing a vectorized floorplan from an RGB image is a challenge that is discussed in [18] in which an existing method "Raster-to-Vector" (RtV) is used for this purpose.

The first step of the RtV method based on a CNN architecture. It takes the RGB floorplan as input and produces as output: 1) wall junctions and 2) pixelwise room classification scores (whether a room is a bedroom, kitchen etc.).

The second step of the RtV method used Integer Programming (constraint programming) to produces a set of 'primitives' based on the junctions from the previous step. Integer programming is used, because floor plans must obey some basic constraints (a room is enclosed by walls, no holes to the outside world except for doors, etc.) These 'primitives' represent spatial structures such as walls, doors and bathtubs. The last step of the RtV method is a post processing step that combines the primitives and pixelwise room classification scores to produce the final vectorized floorplan. The entire process is summarized in the figure below.

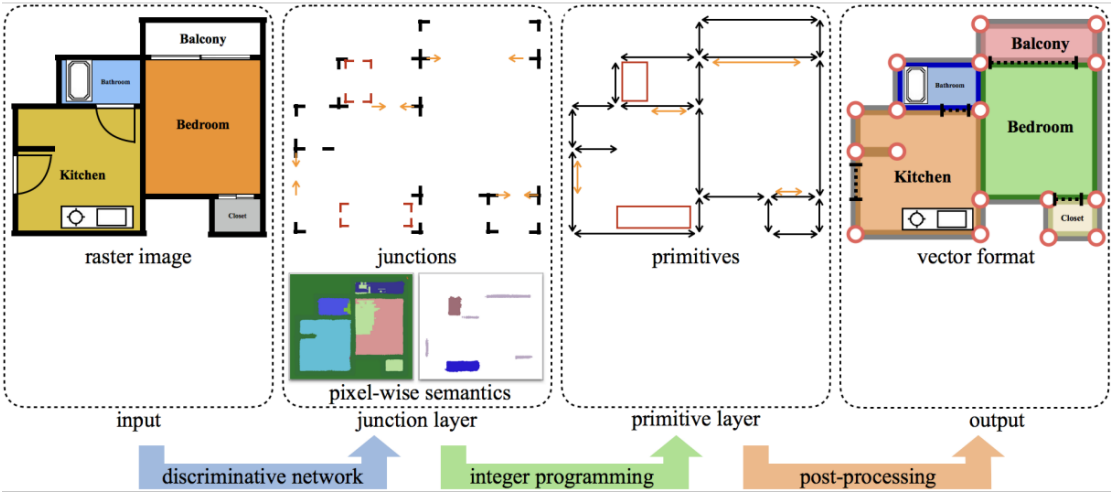


Fig.15 A visualization of the "Raster-to-Vector" method. The RGB image is converted to two intermediate representations which encode structures like walls and doors, as well as room type information. The final step combines this information into a vectorized floorplan.

4.2.2 Constructing a bubble diagram

The process of converting a vectorized floorplan to a bubble diagram is much easier. The vectorized floorplan already contains the information about which rooms exist and their room type. These form the nodes of the bubble diagram. The edges between the nodes are based on the proximity of the rooms in terms of their Manhattan distance. If any part of a room is within 8 pixels of another room, the rooms are connected and thus they have an edge in the bubble diagram. Note that House-GAN, disregards doors – to simplify the problem.

4.3 Data post-processing

4.3.1 From bubble diagram to house layout

To generate a house layout as the final output from the bubble diagram, the generator performs the following steps:

A node is created for each room using the bubble diagram.

Each room is initialized as a 128-d noise vector concatenated with 10-d room type vector. This feature vector is stored as a 3D tensor and therefore is called a feature volume. This feature volume is upsampled to a size of (32 x 32 x 16) after applying CONV-MPN layers.

A three-layer CNN converts this feature volume into a room segmentation mask of size (32×32×1) for each room.

At test time, the room mask (an output of tanh activation function with the range [-1, 1]) is thresholder at 0.0, and the tightest axis-aligned rectangle is generated for each room. Two rooms with an edge must be spatially adjacent with a Manhattan distance of less than 8 pixels.

This completes the generation of the house layout.

4.3.2 The tightest fit axis-aligned rectangle

An axis-aligned rectangle is a rectangle in the Euclidean plane whose sides are parallel to the coordinate axes. An example is provided in Figure 1.

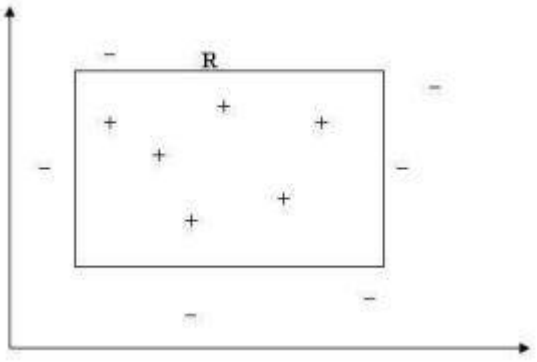


Fig.16 The target rectangle R along with some positive and negative points. This rectangle should fit such that all positive points lie within the rectangle and the negative points lie outside.

A tightest fit rectangle is simply the smallest rectangle that can be drawn for a given set of positive data points as shown in Figure 17

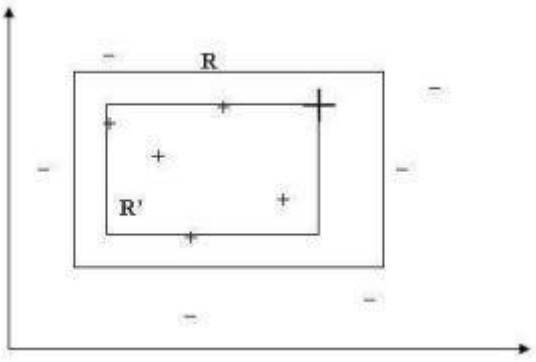


Fig.17 The tightest-fit rectangle R' over the given examples.

In case of the house layout images, each image contains multiple rooms. Each room is represented with a different segmentation mask, all the same size. It is required to fit the smallest axis-aligned rectangle parallel to the image coordinate axis for each room. Once the rectangles are fitted, the rectangles are superimposed to form the house layout image.

Fig.18 Rectangle fitting on the room segmentation mask

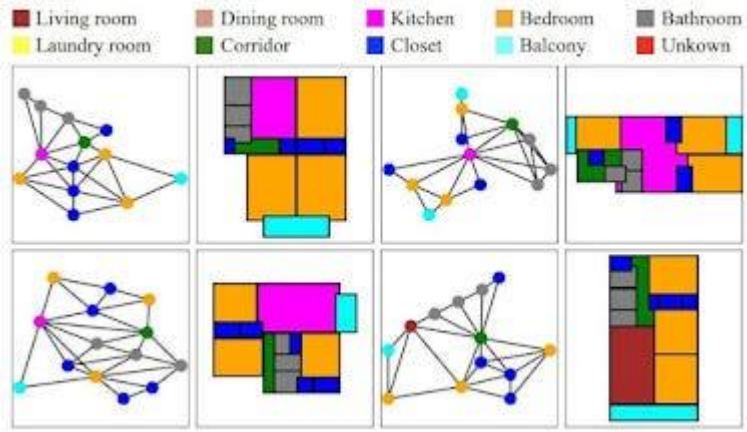


Fig.19 Bubble diagram to house layout

4.4 House-GAN model

In this section the steps and architecture of the generator and discriminator are described in more detail. As a running example we use the following bubble diagram:

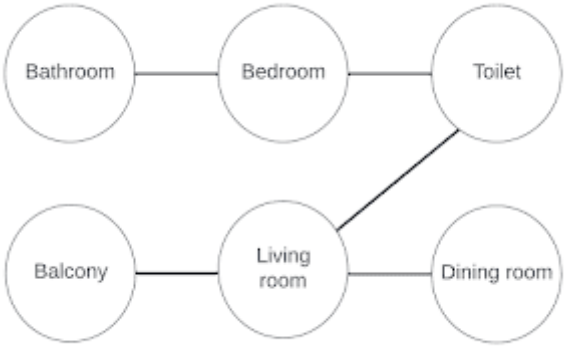


Fig.20 An example of bubble diagram

4.4.1 ConvMPN module

Both the generator and discriminator use ConvMPN modules. In a ConvMPN module a node is updated via convolutional message passing. More precisely, a sum-pools across rooms that are connected to the current node and a sum-pool across rooms that are not connected to the current node are regarded. This results via two features which we concatenate with the previous node feature and then pass to a CNN. The output of this CNN has the same dimensions as the original node feature and forms the new node feature. This process is done for each node in the graph.

$$x_{\text{Living room}} = \text{CNN}[x_{\text{Living room}}; \text{Pool}_{s \in N(\text{Living room})} x_s; \text{Pool}_{s \in \bar{N}(\text{Living room})} x_s]$$

Fig.21 An example of the ConvMPN module for the living room node is given in the figure.

4.4.2 Generator

The generator receives a bubble diagram as input and outputs a graph of room segmentation masks. A schematic overview is given in the figure below.

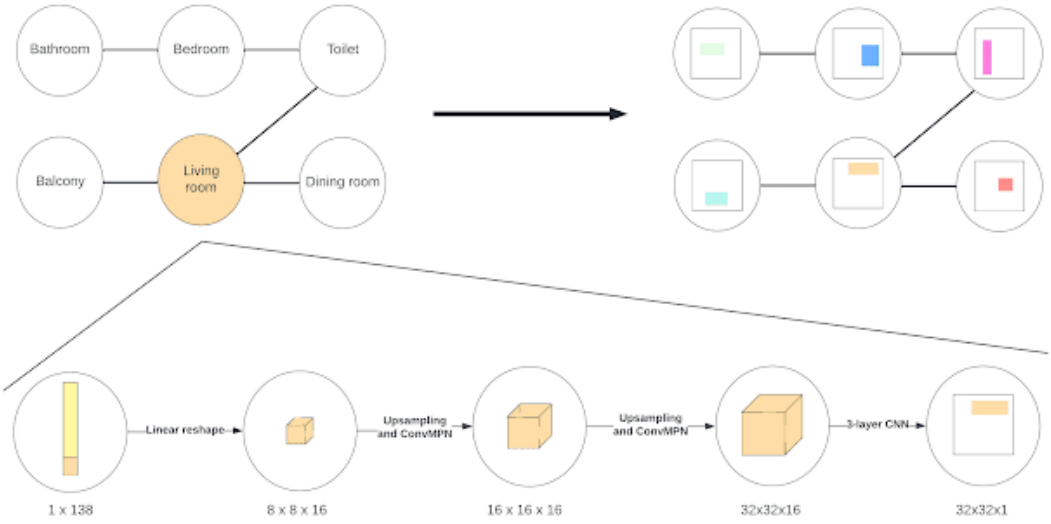


Fig.22 Generator

The process starts with a 138-d vector for every room (i.e. node). The first 128 entries of this vector are random noise sampled from a normal distribution and the last 10 entries represent the room type (one-hot encoded). A linear reshape is done to expanding this vector into a (8x8x16) feature volume. On this feature volume we apply upsampling and a ConvMPN module twice, which results in a (32x32x16) feature volume. As a last step a 3-layer CNN converts this feature volume in the desired (32x32x1) segmentation mask.

4.4.3 Discriminator

The input for the discriminator is the same as the output of the generator, a graph of room segmentation masks. To embed room type information to this segmentation mask a one-hot encoded room type vector is passed through a linear layer, reshaped to (32x32x8) and then concatenated with the segmentation mask. Next a CNN with three layers is applied, which outputs a (32x32x16) feature. After this a downsample and ConvMPN module is applied twice. These two steps can be seen as a reversed version of the upsampling and ConvMPN module in the generator. Lastly, a three layer CNN is applied again to end up with a 128d vector. The result is a vector for every room we sum-pool over all of them and pass the result through a single linear layer. This layer outputs a scalar, classifying real floorplans from fake ones.

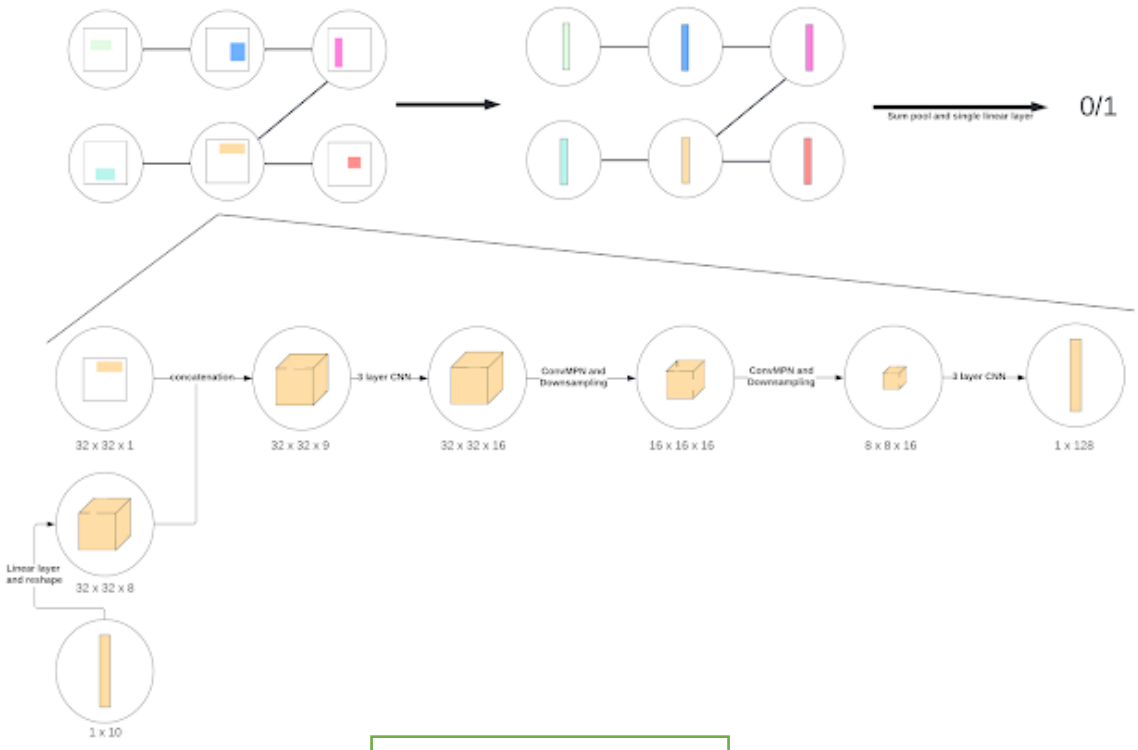


Fig.23 Discriminator

5. Expected Achievements

The project provides a method for converting architectural constraints to graphs and generating a collection of axis-aligned bounding boxes of rooms, which are then translated to floorplans.

The algorithm eliminates the need for the architect's time-consuming and costly iterative process with the customer.

One may expect the algorithm to be able to process a broad collection of realistic house layouts within the bubble diagram constraint, which will be transformed into a realistic floorplan by the architect.

The algorithm's performance will be assessed using realism, variety, and compatibility measures.

6. Research Process

6.1 The process

The project is divided into two phases: the first is research, and the second is implementation. The waterfall methodology was used, requires undertaking considerable study and documentation before moving on to system development.

6.1.1 Phase A

To begin, the project's steps were discussed with our mentors to develop a project plan that outlined the various tasks needed to be achieved during the project. This enabled us to stay organized and on track. Then, relevant papers were reviewed and discussed with the mentors to gain a comprehensive understanding of the related topics, including how neural networks work and train, what layers they are composed of, and what activation functions we use.

Later, CNN , GAN and CONV-MPN were explored aiming to gain a better understanding of the proposed algorithm. Each topic was summarized to use the presentation later. We then consulted the project's supervisors to ensure understanding prior reading the paper again with a focus on the general architecture and workflow of the model. Later, the writing of the project book continued, delving deeper into the generator, discriminator, and W-GAN loss function architectures. Simultaneously, insights and summaries for these topics were added to the project book. Next, the implementation the proposed algorithm in phase B was discussed. The system's GUI, use-case and class diagrams were created along with planning tests that would be run on the system on the next phase. Finally, once we were satisfied with our project book, we began working on the presentation.

6.1.2 Phase B

We embarked on a challenging journey aiming to implement HOUSE-GAN for our study, having high expectations from the model. The research containing git repository for trained model and pretrained model . We decide to go deeply into the pretrained model hoping that the model training will be a base for a following research. While starting running the pretrained model, several obstacles arose. Certain portions of the code that were written in a "hard coded" way needed to be modified (file paths, for example).
In addition, several of the imported packages in the Python code were deprecated or from old versions, while Google Colab is using the most recent versions by default.
That is, versions needed to be downgrade each time until it worked, and "play" with python

packages in order to be adapted to the code (for example, configure Google Collab to work with Python 3.8 by default).

Finally, after running the code successfully, the process seemed proper, without any errors, and generated images using our selected dataset.

6.2 Pre-Trained process

6.2.1 Pre-Trained setup

Architecture : The proposed architecture is based on WGAN-GP (Wasserstein Generative Adversarial Network with Gradient Penalty). It consists of a generator and a discriminator.

Hardware: The model was trained on a workstation with dual Xeon CPUs and dual NVIDIA Titan RTX GPUs.

optimizer :ADAM optimizer is used with parameters **b1=0.5 b2=0.999.**these values balance between adaptability (through quick updates based on recent gradients) and stability (by smoothing the updates with respect to variance). They are often considered as reasonable defaults and are widely used in deep learning frameworks like PyTorch and TensorFlow)

Training Duration**:** The model is trained for 200,000 iterations**.**

Learning Rates**:** The learning rate for the generator is 0.0001, and for the discriminator, it's also 0.0001.

Batch Size**:** The batch size used during training is 32.

(A batch size of 32 strikes a good balance between stability, computational efficiency, and generalization, while also being practical given the memory constraints of GPUs.)

Number of Critics**:** The number of critics (or discriminator updates) per generator update is set to 1**.**

Activation Functions: Leaky ReLU with a slope of 0.1 is used for all non-linearities except for the last layer of the generator, where hyperbolic tangent activation function is used.

Normalization Techniques: Spectral normalization in the convolution layers and a per-room discriminator were experimented with but not utilized in the final model as they did not result in significant improvements.

6.2.2 Pre-Trained Dataset

LIFULL HOME's database offers five million real floorplans, from which we retrieved 117,587. The database does not contain bubble diagrams. The floorplan vectorization algorithm [1] was used to generate the vector-graphics format, later converted into room bounding boxes and bubble diagrams. The vectorized floorplans utilized in this project can be found here, this dataset does not include the original RGB images from LIFULL dataset. In this project dataset FloorplanGraphDataset class was used for working with floorplan graph data including loading, preprocessing, augmentation, and batching.

6.2.3 Pre-Trained Flow

1. Setup and Imports: Importing necessary libraries and setting up some configurations using command-line arguments.
2. Model Loading: Loads a pre-trained generator model from a checkpoint.
3. Dataset Initialization: The script initializes the dataset for evaluation using the FloorplanGraphDataset class.
4. Visualization and Generation: It iterates over the dataset, generates floorplans using the pre-trained generator, and visualizes the generated floorplans along with their corresponding ground truth graphs and masks.

5. Vectorization and Image Generation: The script vectorizes the floorplans, draws graphs, and reconstructs floorplan images. It then saves the generated images and graphs.
6. Final Image Saving: After processing a certain number of images, it saves the generated images in batches.

6.3 The product

As previously stated, the implementation of the algorithm is generate realistic house layouts based on architectural limitations provided as a bubble diagram – the input graph. As a result, it was decided to show how to create a realistic layout that can be transformed to a floorplan using digital architectural tools. Use case and class diagram were constructed, to demonstrate how the model can generate house layouts based on the bubble diagram which is built as input by the user.

6.4 Results

The first column of the table 24 reflects the bubble graph input and four options for floorplan. Room masks can be observed left to each floorplan .

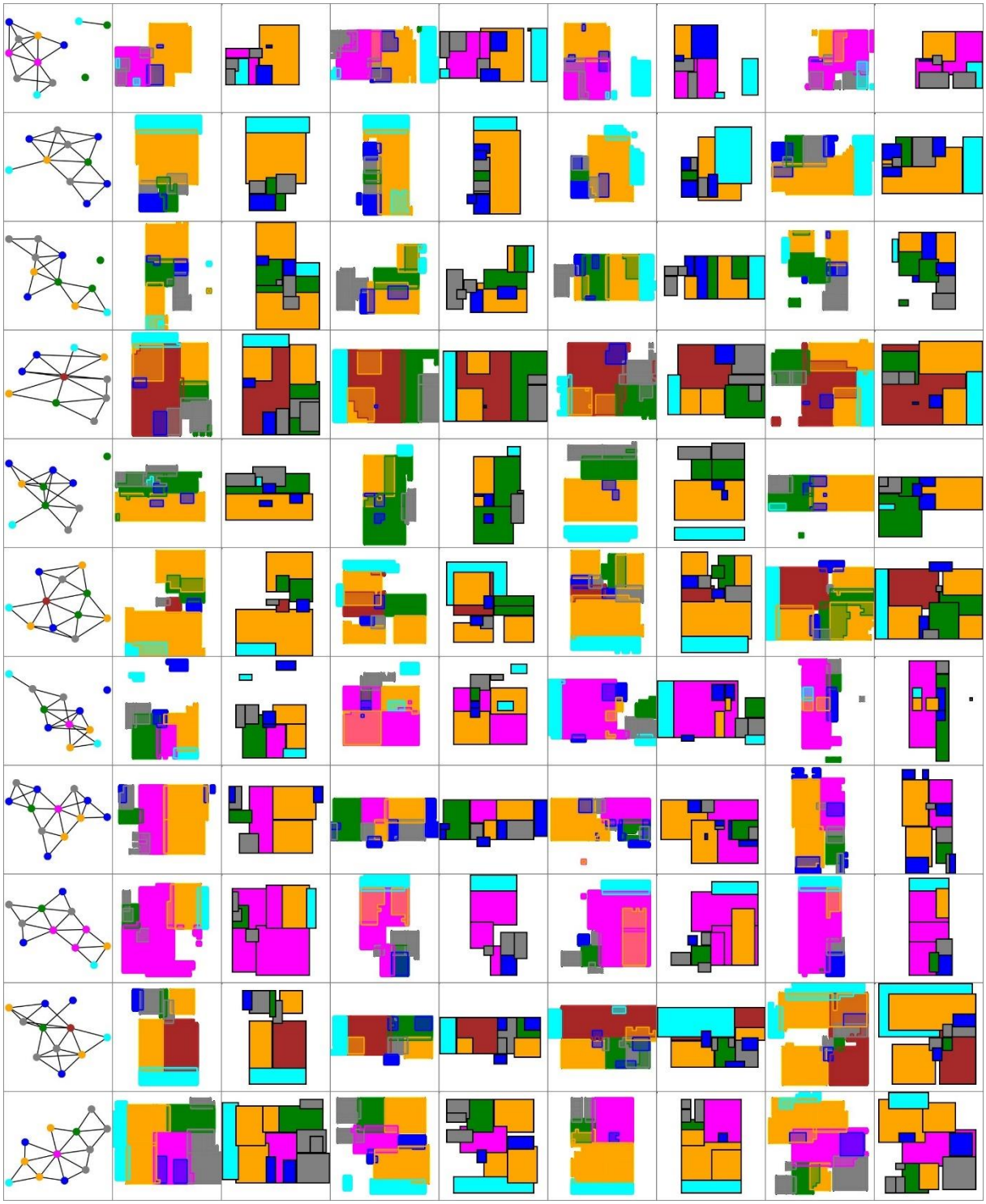


Fig.24: example of output

6.4.1 Failure cases:

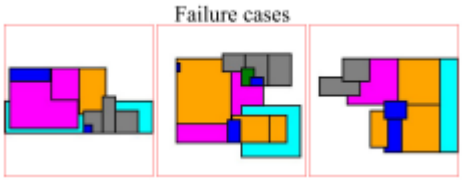


Fig.25: example of failure cases

Figure 25 shows interesting failure and success examples that were compared against the ground-truth in our user study.

Professional architects three failure cases as “worse” against the ground-truth. For instance, the first failure example looks strange because a balcony is reachable only through bathrooms, and a closet is inside a kitchen. The second failure example looks strange, because a kitchen is separated into two disconnected spaces. The considered

major failure modes are

1) improper room size or shapes for a given room type (e.g., a bathroom is too big).

2) misalignment of rooms.

3) inaccessible rooms (e.g., room entry is blocked by closets).

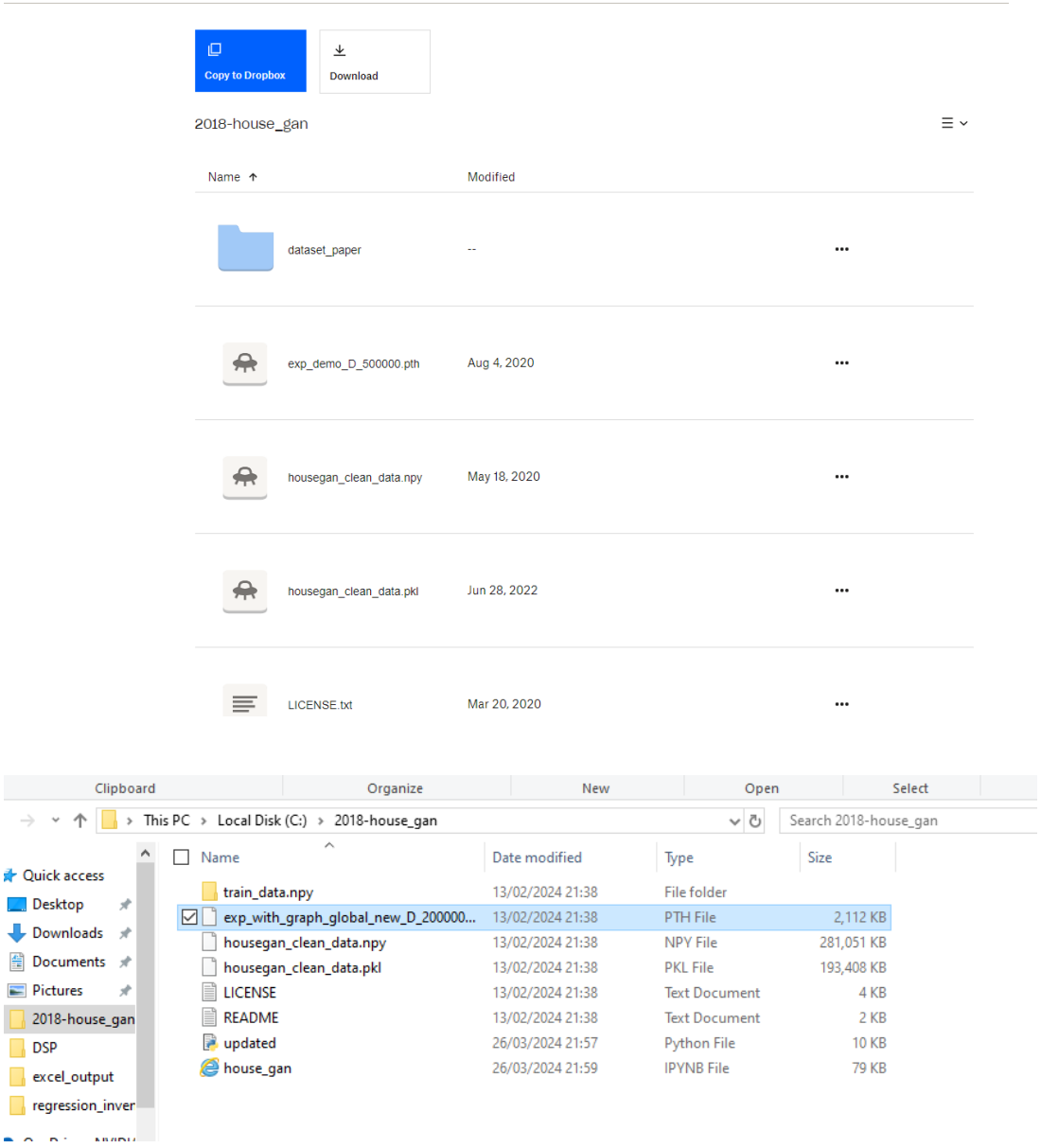
7. User's Guide Operating Instructions

1. Go to code repo on GitHub site: https://github.com/ennauata/housegan

2. In README file go to “running pretrained model” instruction.

In addition to these instructions, there are some more steps to execute / modify for running the code successfully:

3. After downloading dataset, rename “exp_demo_500000.pth” to “exp_with_graph_global_new_D_200000.pth” (as actually appears in code)

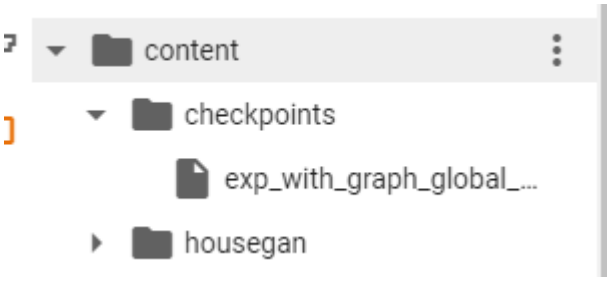


4. Open new notebook in google colab.

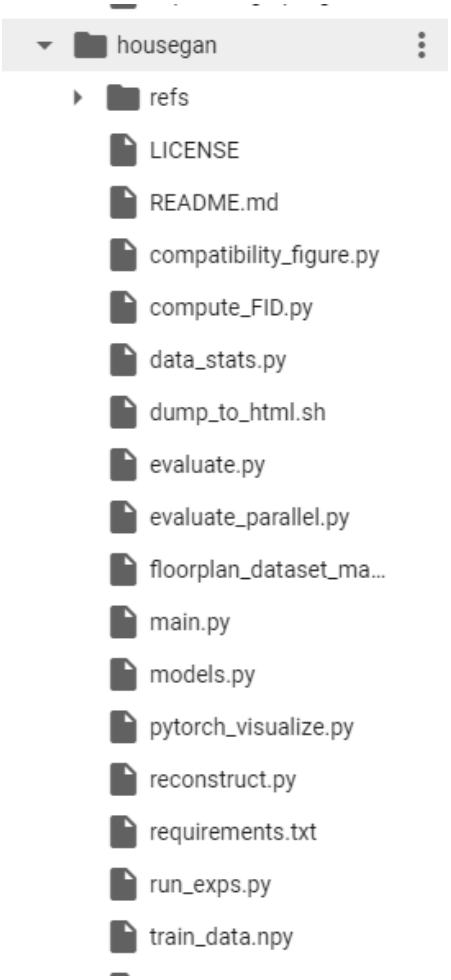
5. Import the project content by the following command**: "!git clone** https://github.com/ennauata/housegan.git**"**

Make sure you can see it in the notebook content:

6. Create "checkpoints" folder under "content" folder, and upload the renamed file from step 3 into it:



7. Upload "train_data.npy" under "housegan" folder (the path will be "/content/housegan/train_data.npy" in colab).



8. Run the following commands on google colab notebook for config appropriate "environment" for running (packages that are required for the project + set python3.8 as the default of colab (old code)):

```
!apt-get update -y
!apt-get install python3.8 python3.8-distutils python3.8-dev
!apt-get install graphviz libgraphviz-dev pkg-config
!wget https://bootstrap.pypa.io/get-pip.py
!python3.8 get-pip.py
!python3.8 -m pip install \
                  torchvision==0.6.1 \
                  Pillow==7.2.0 \
 imageio==2.9.0 \
 networkx==2.4 \
 numpy==1.18.3 \
 opencv-python==4.4.0.42 \
 pygraphviz==1.6 \
 scipy==1.4.1 \
 svgwrite==1.4 \
 webcolors==1.11.1 \
 matplotlib==3.2.1
!update-alternatives --install /usr/bin/python3 python3
/usr/bin/python3.8 1
```

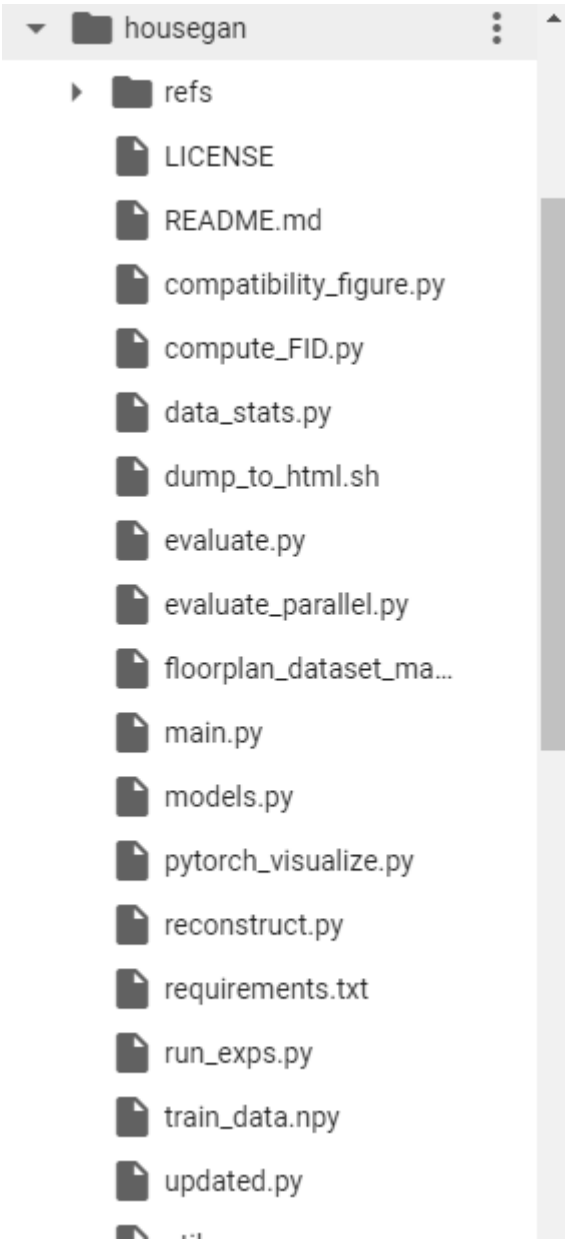
```
!python -m pip install scikit-image

!python -m pip install pycocotools
!python -m pip install ipykernel
!python -m pip install Pillow==7.2.0 \
imageio==2.9.0 \
networkx==2.4 \
numpy==1.18.3 \
opencv-python==4.4.0.42 \
pygraphviz==1.6 \
scikit-image==0.17.2 \
scipy==1.4.1 \
svgwrite==1.4 \
webcolors==1.11.1 \
matplotlib==3.2.1
```

9. Before running " **!*python variation_bbs_with_target_graph_segments_suppl.py***" (README instructions):

- **need to change rooms path in line 180** to "rooms_path = '/content/housegan/' " instead of "rooms_path = '/local-scratch4/nnauata/autodesk/FloorplanDataset/'"

- **need to change line 174 to** "generator.load_state_dict(torch.load(checkpoint, map_location=torch.device('cpu')))"

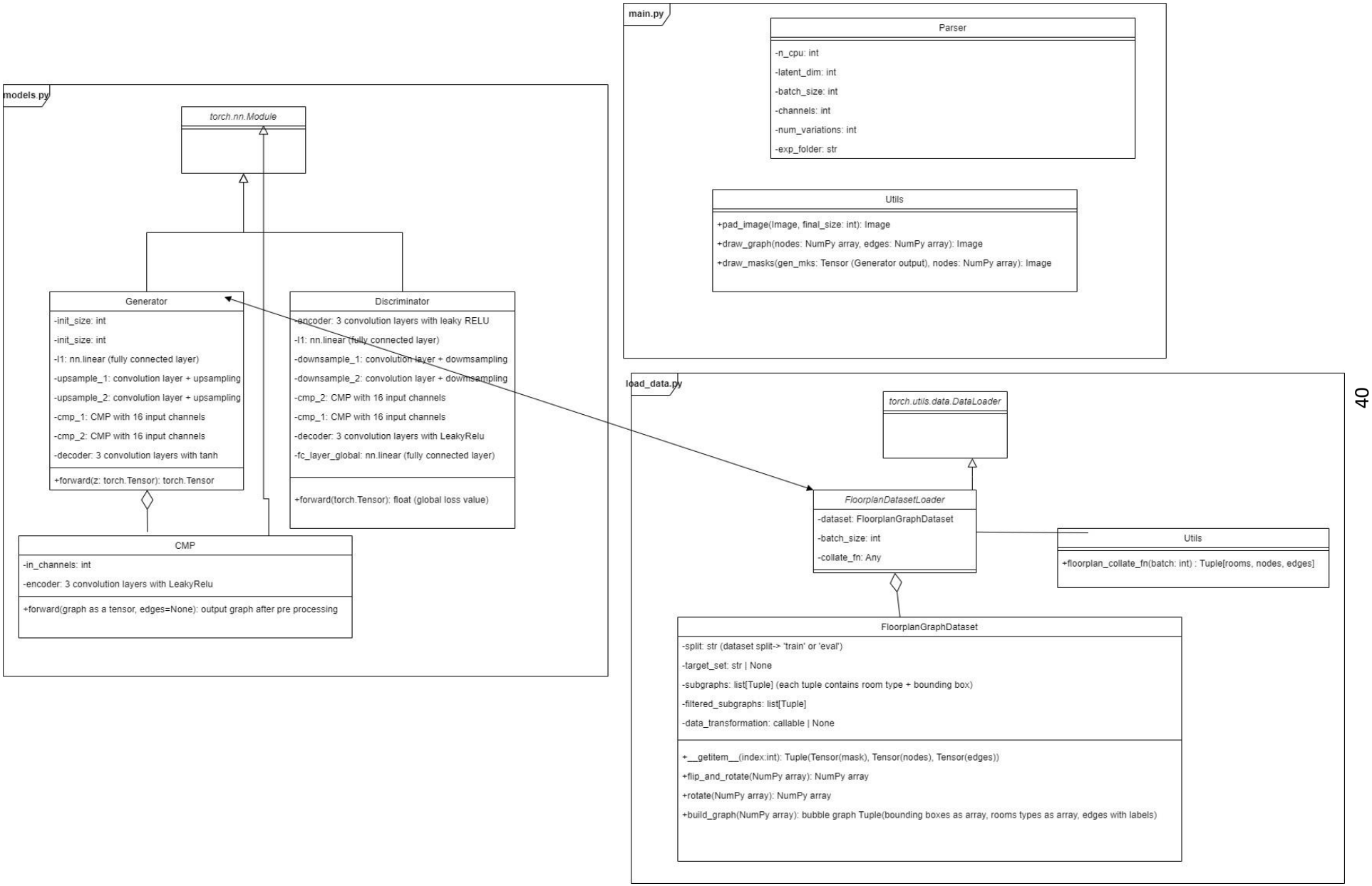
Edit with notepad, save file and upload it to colab under "housegan" (you should use different name for avoid confusing between files, for example : save as "updated")



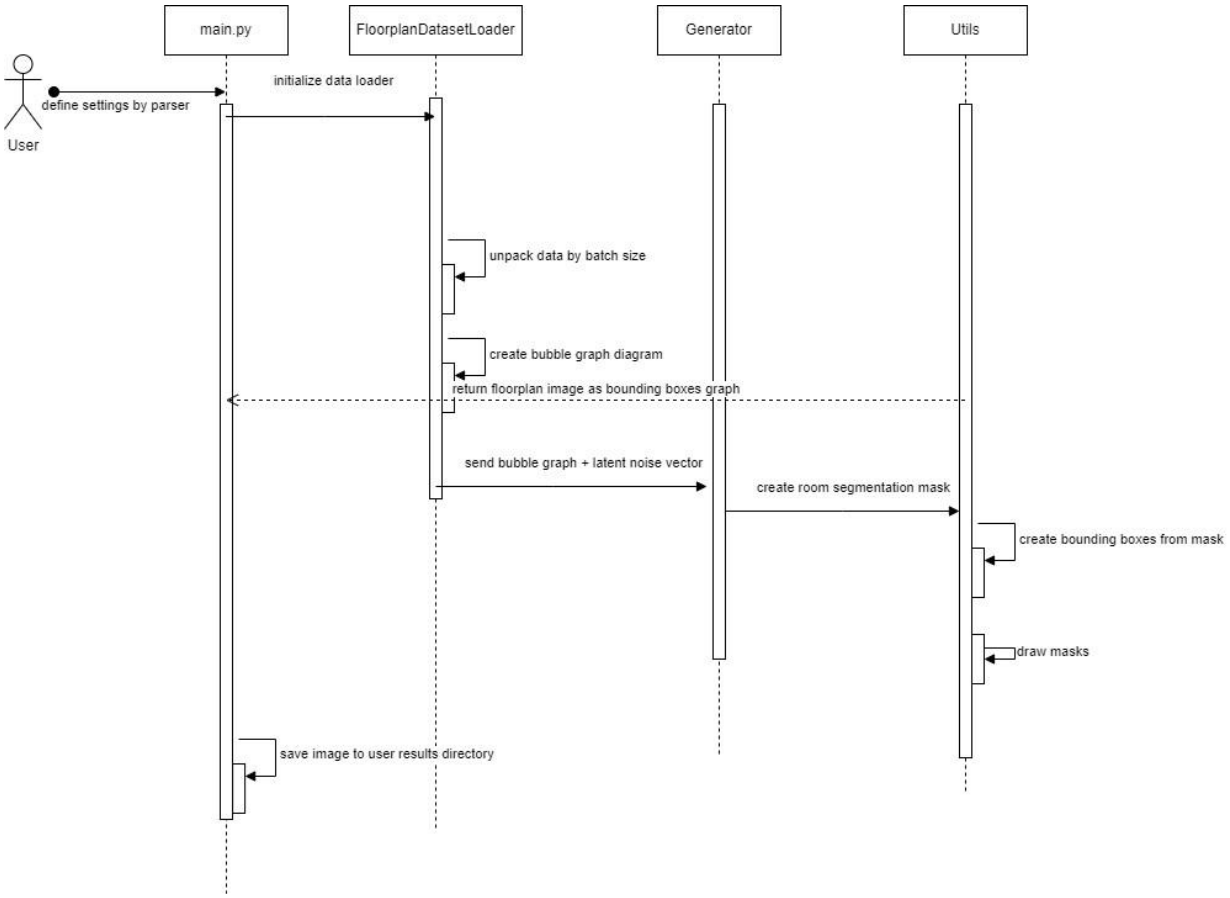
10. Run 😊

```
!python /content/housegan/updated.py
```

8. class Diagram


models.py
torch.nn.Module
Generator
-init_size: int
-init_size: int
-l1: nn.linear (fully connected layer)
-upsample_1: convolution layer + upsampling
-upsample_2: convolution layer + upsampling
-cmp_1: CMP with 16 input channels
-cmp_2: CMP with 16 input channels
-decoder: 3 convolution layers with tanh
+forward(z: torch.Tensor): torch.Tensor
Discriminator
-encoder: 3 convolution layers with leaky RELU
-l1: nn.linear (fully connected layer)
-downsample_1: convolution layer + dowmsampling
-downsample_2: convolution layer + downsampling
-cmp_2: CMP with 16 input channels
-cmp_1: CMP with 16 input channels
-decoder: 3 convolution layers with LeakyRelu
-fc_layer_global: nn.linear (fully connected layer)
+forward(torch.Tensor): float (global loss value)
CMP
-in_channels: int
-encoder: 3 convolution layers with LeakyRelu
+forward(graph as a tensor, edges=None): output graph after pre processing
main.py
Parser
-n_cpu: int
-latent_dim: int
-batch_size: int
-channels: int
-num_variations: int
-exp_folder: str
Utils
+pad_image(Image, final_size: int): Image
+draw_graph(nodes: NumPy array, edges: NumPy array): Image
+draw_masks(gen_mks: Tensor (Generator output), nodes: NumPy array): Image
load_data.py
torch.utils.data.DataLoader
FloorplanDatasetLoader
-dataset: FloorplanGraphDataset
-batch_size: int
-collate_fn: Any
Utils
+floorplan_collate_fn(batch: int) : Tuple[rooms, nodes, edges]
FloorplanGraphDataset
-split: str (dataset split-> 'train' or 'eval')
-target_set: str | None
-subgraphs: list[Tuple] (each tuple contains room type + bounding box)
-filtered_subgraphs: list[Tuple]
-data_transformation: callable | None
+__getitem__(index:int): Tuple(Tensor(mask), Tensor(nodes), Tensor(edges))
+flip_and_rotate(NumPy array): NumPy array
+rotate(NumPy array): NumPy array
+build_graph(NumPy array): bubble graph Tuple(bounding boxes as array, rooms types as array, edges with labels)

9.Sequence Diagram



10. Verification Plan

We tested the following tests to evaluate the system performance and stability:

Test number	Test subject	Expected result
1	In the system's train window, upload a data set that is not a floorplan images dataset.	Invalid input error message.
2	Click "Build Bubble Diagram" button.	A new window is opened for build a new bubble diagram by room types and their spatial adjacency.
3	Click "Generate Layout" button.	A new window is opened and shows the loading of the generation process.
4	In the system's train window, enter a non-integer number in the fields epochs, learning rate and loss evaluations.	An error message indicating an input type mismatch appears.
5	Click "Reconstruct" button on the generation result window.	A new house layout is generated.
6	Click "Download" button on the generation result window.	The house layout is saved on the computer.

Fig.26: test plan table

11. Summary

As technology advances in our environment, it would be a waste not to use its benefits to speed up operations and avoid wasting money.

The production of a house layout is a good example of this; during the project, the issue of the process of planning and building a house is clearly optimized by using the process of producing the house layout by the model which without the model is entirely dependent on the architect and takes a significant amount of time and money.

Designing a house layout using the HouseGAN model is explicitly discussed as well as several ways for creating house layouts used in the past, various and complicated architectures, and functionalities.

Throughout the research, we were impressed by how exciting and influential the area of machine learning is, especially when applied to civil engineering rather than just computer science.

12. GitHub Repository

https://github.com/yael918/HOUSEGAN

13. References

[1] Bao, F., Yan, D.M., Mitra, N.J., Wonka, P.: Generating and exploring good building layouts. ACM Trans. Graph. (TOG) 32(4), 1–10 (2013)

[2] Harada, M., Witkin, A., Baraff, D.: Interactive physically based manipulation of discrete/continuous models. In: Proceedings of the 22nd Annual Conference on Computer Gand Interactive Techniques, pp. 199–208 (1995)

[3] M¨uller, P., Wonka, P., Haegler, S., Ulmer, A., Van Gool, L.: Procedural modeling of buildings. In: ACM SIGGRAPH 2006 Papers, pp. 614–623 (2006)

[4] Peng, C.H., Yang, Y.L., Wonka, P.: Computing layouts with deformable templates. ACM Trans. Graph. (TOG) 33(4), 1–11 (2014)

[5] Hendrikx, M., Meijer, S., Van Der Velden, J., Iosup, A.: Procedural content genreration for games: a survey. ACM Trans. Multimedia Comput. Commun. Appl. (TOMM) 9(1), 1–22 (2013)

[6] Ma, C., Vining, N., Lefebvre, S., Sheffer, A.: Game level layout from design specification. Comput. Graph. Forum 33, 95–104 (2014). Wiley Online Library

[7] Wuang, K., Savva, M., Chang, A.X., Ritchie, D.: Deep convolutional priors for indoor scene synthesis. ACM Trans. Graph. (TOG) 37(4), 1–14 (2018)

[8] Ritchie, D., Wang, K., Lin, Y.A.: Fast and flexible indoor scene synthesis via deep convolutional generative models. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 6182–6190 (2019)

[9] Wu, W., Fu, X.M., Tang, R., Wang, Y., Qi, Y.H., Liu, L.: Data-driven interior plan generation for residential buildings. ACM Trans. Graph. (TOG) 38(6), 1–12 (2019)

[10] Jyothi, A.A., Durand, T., He, J., Sigal, L., Mori, G.: LayoutVAE: stochastic scene layout generation from a label set. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 9895–9904 (2019)

[11] Li, J., Yang, J., Hertzmann, A., Zhang, J., Xu, T.: LayoutGAN: generating graphic layouts with wireframe discriminators. arXiv preprint arXiv:1901.06767 (2019)

[12] Wang, K., Lin, Y.A., Weissmann, B., Savva, M., Chang, A.X., Ritchie, D.: Planit: planning and instantiating indoor scenes with relation graph and spatial prior networks. ACM Trans. Graph. (TOG) 38(4), 132 (2019)

[13] Merrell, P., Schkufza, E., Koltun, V.: Computer-generated residential building layouts. ACM Trans. Graph. (TOG) 29, 181 (2010). ACM

[14] Johnson, J., Gupta, A., Fei-Fei, L.: Image generation from scene graphs. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1219–1228 (2018)

[15] Ashual, O., Wolf, L.: Specifying object attributes and relations in interactive scene generation. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4561–4569 (2019)

[16] Zhang, F., Nauata, N., Furukawa, Y.: Conv-MPN: convolutional message passing neural network for structured outdoor architecture reconstruction. arXiv preprint arXiv:1912.01756 (2019)

[17] Lifull home's dataset. https://www.nii.ac.jp/dsc/idr/lifull

[18] Liu, C., Wu, J., Kohli, P., Furukawa, Y.: Raster-to-vector: revisiting floorplan transformation. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2195–2203 (2017)