CSCI 1430 Final Project Report: The Architectural Illustrator

Experimental Subject Group 4: David Heffren, Yizhou Tan
Brown University

Abstract

This project implements a Conditional GAN to take in the edges of architectural structures and produce an image reflecting a photograph of what the actual structure might look like. It uses the Pix2Pix model, essentially a Conditional GAN with a UNET generator and a PatchGAN discriminator to learn how to generate realistic-looking images of buildings. We were able to get decent results with a limited dataset, with the model learning shadows without prior knowledge, and correcly filling out most of the colors of the image, although failing to replicate the more unusual color palettes. However, in this regard it does similar to how a human would do on guessing the colors of a building, and although the output is a bit blurry, it's still quite realistic.

1. Introduction

The problem we wish to solve is to go from a line drawing of an architectural design to what the actual building would look like. We're essentially filling in the details of the design, including color, texture, and features, from just the edges themselves. This is a difficult problem because if you were to look at the edges of a building, there's not really an intuitive way of knowing what color it will be, or where to fill in the details. This is essentially the jobs of the architects and designers, to go from their sketch of the project to the finished product. We're essentially working with limited information.

To solve this problem, we will be using a Generative model to generate images of the real architecture conditioned on the outline and edges of the image. There are 2 main reasons why we chose a generative model as opposed to just a straightforward CNN to go from outline to the original image. First, we might want to introduce some randomness into the equation, as we want the generated images to be creative renditions of the outline,rather than specifically carbon copies of the original. The point of the lack of information is to experiment and find many different potential designs which may be feasible, and a Generative model accounts for this randomness. In addition, the learning objective of Generative models, specifically that of the GAN (which we will be implementing) allow for a more high-level interpretation of what a ”good” generated image looks like. By defining a good generative image as ”one that is indistinguishable from reality”, it helps us avoid making assumptions in a loss function which would narrow the scope of that objective.

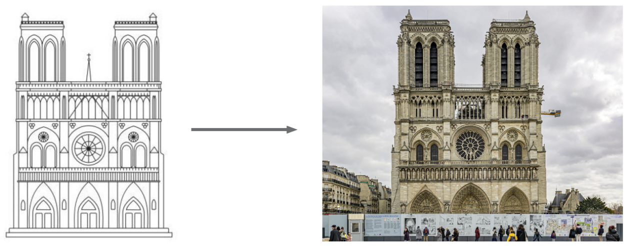
Figure 1. Conceptual Problem

The application of this solution is quite clear, in that it would help architects go from a sketch or prototype of a building or structure to get a potential visualization of what it would look like. Of course, edge detection isn't exactly the same thing as architectural sketches, but this would be a starting point to look at how to go from a simplified rendition of an idea to an actualization of that idea. Other than helping architects quickly visualize their ideas, this also fosters the community engagement aspect of architectural and urban design. It enables people without an architectural background to come in and easily suggest what kinds of building they want in a neighborhood.

2. Related Work

For this project, we used a Conditional Generative Adversarial Network, introduced in Goodfellow et al. [2] and Mirza et al. [4]. Specifically, we implemented the architecture designed for Image-to-image translation introduced in Isola et al. [3], and applied it to our specific brand of problem. This architecture uses a U-net structure for the generator, introduced in Ronneberger et al. [5]; and uses a PatchGAN structure for the discriminator, which is a form a texture/style loss [3]. We used Tensorflow to create our model, and we used Google Colab to run it.

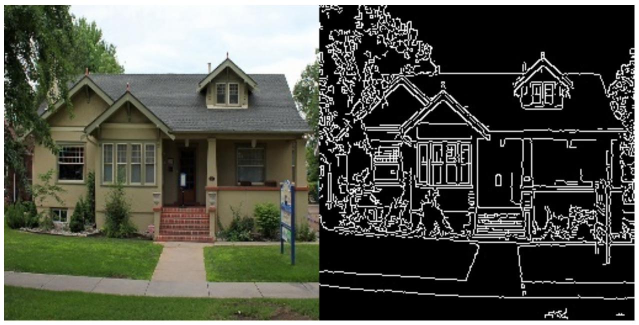
Figure 2. Example Data and Preprocessing Result

3. Method

3.1. Dataset and preprocessing

For our dataset, we used the dataset introduced in this paper [6], which contains 25 classes of architecture and around 4,700 images total. We loaded these images(using multiprocessing), then resized them as 256x256 images as done in Isola et al. [3].

Then, to get the edges, we used a Canny edge detector with parameters 100 and 200, and a filter size of 3. This worked pretty well, although it definitely had more detail than was ideal. We then extracted a subset of about 800 of these images to use, since the full dataset caused our program to run out of memory. The ones we chose were generally quite similar, as some of the architectural styles were so distinct they messed up results (such as the Achmenaeid architecture). We split them into train and validation data by a 80x20 split.

3.2. Architecture

For this problem, we implemented a conditional GAN, which consists of 2 separate models trained together: A Generator (G) and a Discriminator (D). The generator maps a condition y and latent noise vector z to a generated image $\hat{x}$. The discriminator takes in the same condition y and an image x, and classifies it as either real or fake. In our application, the conditional input would be the edges/outline of the image, and the generated image would try to replicate the original image that the outline represented.

The objective of the conditional GAN is expressed as

$$Ł_{CGAN}(G,D) = \mathbf{E}_{x,y}[logD(x,y)] + \mathbf{E}_{y,z}[log(1-D(G(x,z))]$$

where G tries to minimize this objective against an adversarial D that tries to maximize it. The intuition of this is that the Discriminator wishes to maximize the log likelihood of classifying all images correctly, whereas the Generator wants to maximize the log likelihood that the Discriminator classifies the fake images as real (ie it tricks the discriminator). Note that as suggested in the GAN paper [2], we wish to minimize the negative log likelihood instead. we may also add an L1 loss component to the generator, to ensure it stays close to the real image.

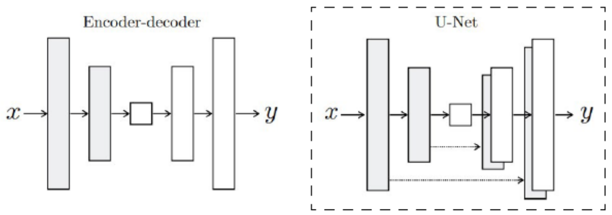

Figure 3. Difference between Encoder-Decoder and U-Net [3]

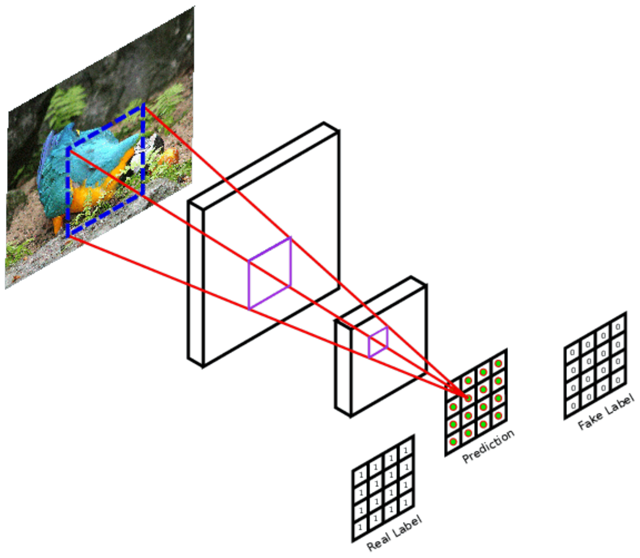

Figure 4. PatchGAN Discriminator [1]

As for the architecture of the generator and discriminator, we use the designs suggested in the paper [3]. For the Generator, we use the U-Net architecture [5], which is an encoder-decoder architecture with skip connections between encoder and decoder layers. The goal of this architecture is to use convolutional layers to downsample and extract the important information from the input, encoding its properties in a hidden vector, then using Transpose Convolution to upsample back into an image of the original size, which hopefully learns how the original image would be colored/shaded. The residual connections are intended to give later layers access to the outputs of earlier layers, and avoid the information loss caused by the bottleneck. However, we also implemented our own simpler version of the U-Net, with half the number of parameters, and also tested the vanilla encoder decoder model.

For the architecture of the discriminator, we used the Markovian PatchGAN architecture [3]. Instead of classifying the entire image as real or fake, instead we classify independent ”patches” of the input image as real or fake. Each pixel of the ouptut represents a patch of the input (specifically its receptive field). We’re representing the image as a Markov

Random Field, assuming that pixels in different patches are independent, and that the characteristic of a pixel can be understood only by neighboring pixels. The motivation for this architecture is to better model high-frequency structure, in other words to capture the details of the input image. It can be understood as a form of texture/style loss, looking at high-frequency features rather than low frequency ones (which will get a blurrier result). We chose the size of our receptive fields to be 16x16 (the paper uses 70x70) for more computational efficiency. Thus, our output is size 32x32.

3.3. Optimization

In our trainStep function, we alternated between one gradient descent step on the Discriminator, then one on the Generator, as suggested in the original GAN paper [2]. We also tried doing them both at once, but this led to terrible results, where it wouldn't converge at all. This is most likely due to the fact that since the weights of each are changing without seeing how the other changes, they keep learning irrelevant data (they're learning based on the old way the discriminator/generator behaved) which caused neither to really update appropriately.

For our loss function, we used Binary Crossentropy for both, with opposite labels for the generator and discriminator. We also added the L1 component to the Generator, with a regularizing coefficient of 100. However, I think this overemphasized the L1 component as opposed to the tricking the discriminator component.

We used the Adam optimizer with a learning rate of .0002, $\beta_1 == .5$, and $\beta_2 = .999$. We used a batch size of 4, and trained our model for 40 epochs. Initialized all weights with random normal initializer with mean 0 stdev .02

4. Results

We tested multiple different architectures, losses and styles. We first implemented the U-Net Generator exactly as the paper [3] (see Fig.5 for results). Then, we tested out a simple Encoder-Decoder structure without the U-Net residual connections (see Fig.7). The results were not satisfying because the model did not preserve the information from the edges and the generated images are very blurred. We then switched back to the U-Net, but tested out a different architecture that was about half the size of the original one (see Fig.7 for U-Net II and Appendix 7.1). It had roughly the same performance but was much more cost-effective. Therefore, we tested two other iterations of this model, relatively without the PatchGAN (we used a binary classification dense layer instead) and without the L1 loss. See Fig.7 for results.

As you can see from the images, the best performing model was UNet- I, with U-Net II being a close second, and much more computationally efficient. The colors are close to the original, and it captures a lot of the little details of the edges. You'll notice that for all of the images, architecture

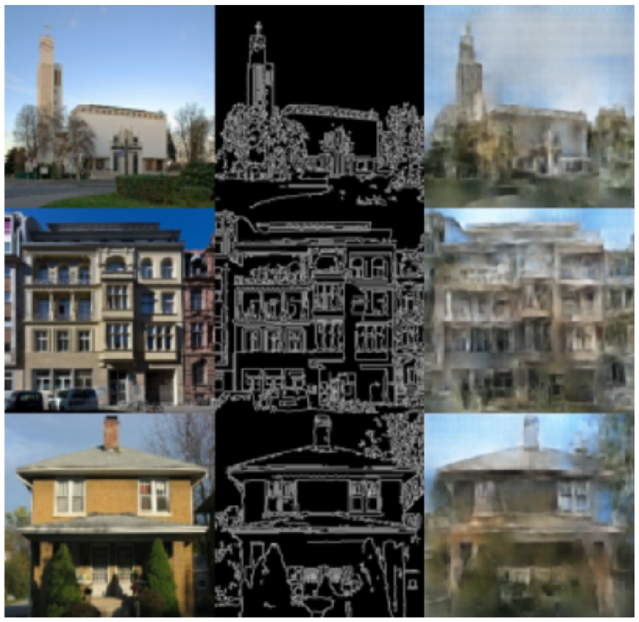

Figure 5. U-Net I Model Output. *Left:* Original Image. *Middle:* Edges. *Right:* Model Output.

which had a "strange" color wouldn't be reflected in the generated image, which is as expected. This is shown in the image of the house with the red roof. Specifically, for the generated images without L1 loss, their colors don't really match up at all, which is understandable as its only objective was to fool the discriminator, not to have matching colors. This model allows for more creativity and expressivity in the outputs, which is beneficial. You can see that without the PatchGAN, the generated images are muddy and unrealistic, demonstrating that the focus on specific high frequency textures is an important one. Since it takes in ALL pixels from the image into account for its classification, it doesn't see a problem with there being blue sky in the middle of a building, whereas the PatchGAN would expressly penalize blue sky which wasn't at the top of the image. Again, the encoder decoder completely failed due to the bottleneck of information at the end of the decoder. Without having access to the original edge information from the encoder, the reconstructed image in the decoder is blurry and unfocused.

5. Technical Discussion

The choice to go from edges to images rather than from an outline/sketch to an image was one that was more practical than ideal, as the point of the project was more to go from an architectural sketch to reality. However, obtaining such a large amount of architectural sketches, due to a lack of premade datasets related to this, the 10+ hours it would take to draw all those sketches by hand, and the likely case that it wouldn't even learn anything due to variation in sketch quality meant that this was unrealistic, thus we decided to

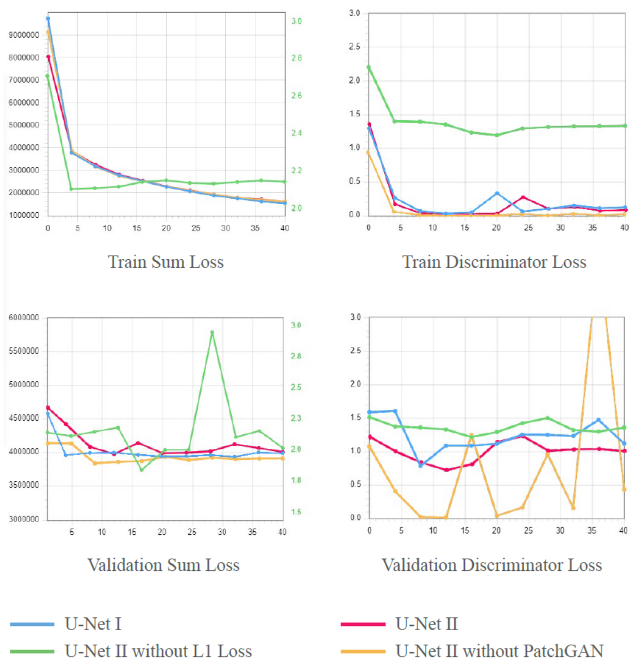

Figure 6. Performances Comparison

go from edges to images instead.

I think that this simplification is an easier version of the other problem, as all it has is more detail. However, it also raises the problem that some images will have edges which don't really get the basic structure of the building, as the edge detection is flawed. For example, many images with trees had way too many lines and edges which weren't really essential to the actual structure of the house we were focused on. Potentially, in the future we could implement a way to go from the sketch TO the edges, and then combine the two models together to go from sketch to image.

Another tradeoff we had to make was regarding the randomness of the model itself. Even though GANS are supposed to take in an input of a noise vector, we decided (as well as the original paper) to not use that, and instead implement randomness through dropout in the generator decoder. However, this really didn't generate much randomness, and we wanted to find a way to get better random results. Randomness wasn't ESSENTIAL to the task, especially since we had so much details in the edges, there wasn't a ton of room for creativity. I feel that in the case we went from sketches to images instead, the randomness would be much more important. Related to this, we would have liked to output a distribution of potential images for each edge, rather than just one explicit image, but we found the task of generating distinct images without this random component to be quite difficult, but with further time we could definitely address it.

6. Social Impacts

This model helps architects quickly visualize their design ideas, and improve design efficiency. While this means less workload for the architects, it might also means smaller number of architects needed in the field. On the other hand, such visualization method enables the community engagement aspect in architectural and urban design. Any citizen or neighbor would be able to use this tool to express vividly what they want for the neighborhood.

7. Conclusion

Using a conditional GAN, we created a visualization tool that translates sketches to realistic architectural renderings. This makes it possible for people, architects or not, to express their design ideas conveniently. This facilitates communication between people and stakeholders throughout the earlier phases of the design process, and specifically, the community engagement process. This project is a manifestation of how artificial intelligence can be used in design.

References

[1] Uğur Demir and Gozde Unal. Patch-based image inpainting with generative adversarial networks. 03 2018. 2

[2] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014. 1, 2, 3

[3] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2016. 1, 2, 3, 6

[4] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014. 1

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 1, 2

[6] Zhe Xu, Dacheng Tao, Ya Zhang, Junjie Wu, and Ah Chung Tsoi. Architectural style classification using multinomial latent logistic regression. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 600–615, Cham, 2014. Springer International Publishing. 2

Appendix

Team contributions

David Heffren What I did for the project was most of the coding and debugging portions of the project, such as finding the dataset, loading it, designing the tensorflow models and functions etc. I read the original papers, then implemented the models based on them and the task at hand. I designed the structure of the code, wrote all the methods regarding loading data, edge detection, the model's train loop, and designing classes like UNET,

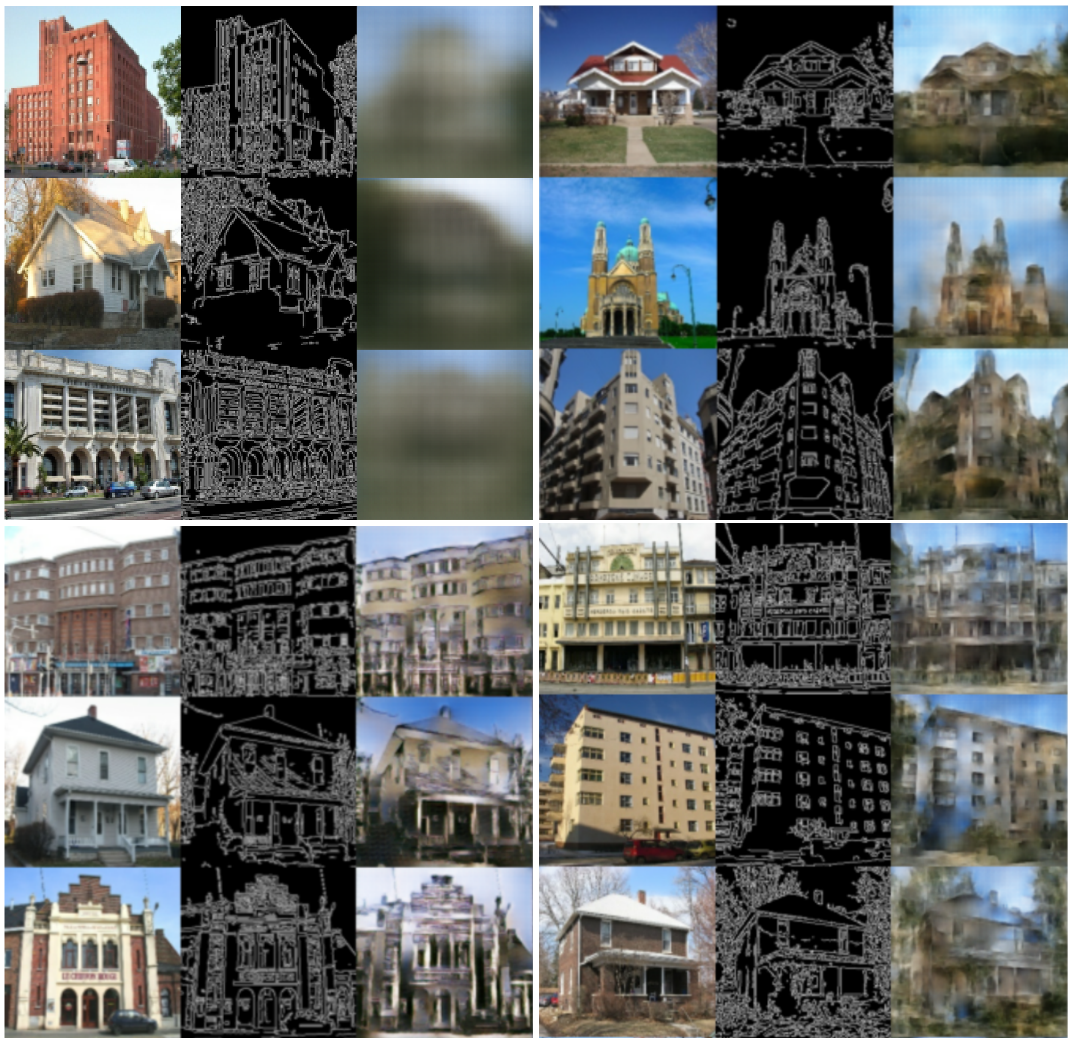

Figure 7. Results. *Top-Left:* Encoder-Decoder. *Top-Right:* U-Net II.*Bottom-Left:* U-Net II without L1 Loss. *Bottom-Right:* U-Net II without PatchGAN.

EncDec, PatchGAN, Discriminator, Generator etc. I also did an example GAN on the MNIST dataset to try to get a hang of what it's like. I used tensorflow to make the proper classes, designed the train loop and fit, made the preprocessing/edge detector, made the method to load the images from their files with multithreading(Extremely difficult, this took like 10 hours and it doubled the speed of downloading), and I ran the first few attempts of the model(without a gpu) debugging it. I also added the callback displaying the images. I wrote about half the report, and helped with the training process throughout.

Yizhou Tan I proposed the question, found theoretical references and proposed datasets and data collection methods. To successfully run the code, I moved it to Colab and rewrote some code to deal with the memory issue of Colab. I rewrote the whole dataset preprocessing method to separate preprocessing from running the model and rewrote the loading method so it loads the data correspondingly. I debugged the code until it ran, and implemented and tested out 6 different variations of the model, 4 of them we ended up including in the report. Variations include different parameters, different model architecture, different loss methods for the generator, and different discriminator types. I collected the results, plotted the graphs to compare performances, wrote the other half of the report and made the poster.

7.1. Specific Architecture

The specific architectures of the GAN were as follows. We structured each model in terms of Convolution Blocks, which were a custom layer consisting of a Convolution layer with a stride of 2 and a kernel of 4x4, a BatchNormalization layer (potentially), a Dropout Layer (potentially), with a

Relu layer. Denote one of these blocks without BN or D as Ck where k is number of filters, denote one with Batch normalization as CBk, and one with Dropout as CDk (CBDk for all 3).

Note that all decoder convolution blocks are actually Convolution Transpose blocks, as they upsample the image by 2, doubling its size each layer.
Our Generator U-Net I has the structure: Encoder: C64 $\rightarrow$ CB128 $\rightarrow$ CB256 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512
Decoder: CBD512 $\rightarrow$ CBD1024 $\rightarrow$ CBD1024 $\rightarrow$ CB1024 $\rightarrow$ CB1024 $\rightarrow$ CB512 $\rightarrow$ CB256 $\rightarrow$ CB128
Each layer in the encoder has Leaky relu with slope .2, an deach layer in the decoder is leaky. However, after the last layer in the decoder, we apply a convolution layer with stride 1 to map to 3 output channels, and apply a tanh activation.
Generator U-Net II:
Encoder: C64 $\rightarrow$ CB128 $\rightarrow$ CB256 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB512
Decoder: CBD512 $\rightarrow$ CBD512 $\rightarrow$ CBD512 $\rightarrow$ CB512 $\rightarrow$ CB512 $\rightarrow$ CB256 $\rightarrow$ CB128 $\rightarrow$ CB128
We changed this from the paper [3] by adding a layer in the decoder and changing the output size of the last layer to better integrate with our U-Net architecture (want same size final as the intitial)

Discriminator Architectures:
PatchGAN 16x16: C64 $\rightarrow$ CB128, leaky relu, then a convolutional layer after to map to 1d. Sigmoid activation function at the end.