Font Generation Using Autoencoders and Discriminator Networks

Andrew Chen, Martin Shi, and Howard Yen

December 2020

1 Abstract

Using a font dataset pulled from Google Fonts, we trained a Deep Convolutional Neural Network, consisting of an autoencoder and a discriminator, that takes in one letter of a certain style and outputs a full alphabet with a consistent, matching style. Unlike many previous font generation models, our model only needs one letter as input, minimizing the amount of work it would take to use the model to generate your own font; other models usually require base images of a full set of characters to perform a style transfer on. In our case, the capital letter R was used to generate the 26 uppercase letters of the English alphabet, but it is possible to extend our model to lowercase letters, numbers, and other characters. We use a combination of L1 loss, SSIM loss, a localized loss, and discriminator loss to evaluate and train the autoencoder, while the discriminator is evaluated and trained with binary cross-entropy loss. The resulting model showed the ability to learn both the style and the shape of the alphabet when given a letter from a specific font type.

2 Introduction

2.1 Motivation

Suppose that you are looking for a font for your new graphic design, publication, or other project. As someone without professional experience in visual art, you're hoping that you don't have to manually draw out all 26 letters in your desired style (or even more when accounting for uppercase and lowercase letters, numbers, and symbols!), not to mention ensure that all the fonts are up to publishable quality. Using an existing font would require you to pay the artist a licensing fee, and even then, existing fonts may not suit your needs. Commissioning a new font would require even more time and money.

Fonts generally require professional designers to create, and the process can often take days or weeks. It's very difficult for a regular person to create their own fonts, especially because it's hard to maintain

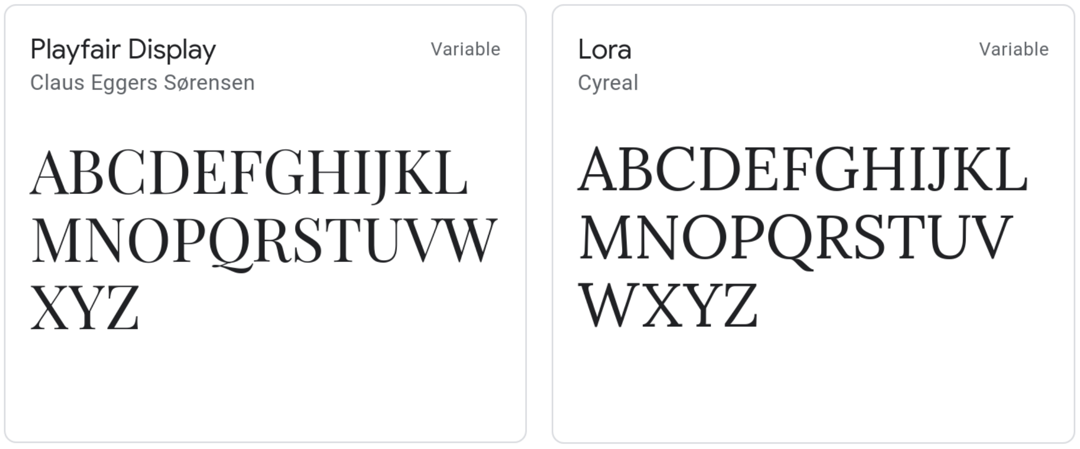

Figure 1: Examples of some fonts downloaded from Google Fonts

a consistent style for every letter. A designer might also want to change a certain aspect of a font, but again, it is difficult to subtly change the style of every single character consistently.

Therefore, our goal for this project is to create a model that is able to generate an entire font set with a consistent style based on just one letter. The end user would be able to input an image of a letter written in their desired style, and the model will generate the rest of the font set from that one image.

2.2 Previous Works

Generative models in computer vision have been explored extensively in the past; we decided to take inspiration from Generative Adversarial Networks (GAN). GANs consist of two essential networks: a generator and a discriminator. The generator takes in a latent vector and generates some results that are supposed to mimic the ground truths in the data set, and the discriminator takes in input from both the original images from the data set as well as the output from the generator, and tries to differentiate between the two [8]. The two networks help each other train: the discriminator becomes increasingly better at differentiating between real and generated/fake images and the generator learns to fool the discriminator as it updates the values/weights in its network using loss calculated from the discriminator. As a result, we end up with a generator is that is capable of generating images that have close resemblances to the images in the original data set, which fits the goal of our project very well.

Research relating to GAN has been rather active. Convolution layers have made their way into GAN in the form of DCGAN (Deep Convolutional Generative Adversarial Networks) and proved to be rather useful [7]. This paper found that replacing pooling with strided convolution layers, using batchnorm in both the generator and the discriminator, using LeakyReLU in the discriminator, and using Tanh in the generator were helpful. As we will discuss in the next section, we took a lot of inspiration from this paper's architecture while designing and implementing our own.

Another generative model that has shown promising results is OpenAI's Image GPT, which achieved results comparable to that of GAN, and even better in some cases [1]. Image GPT took a completely different approach from GAN: it took inspirations from Natural Language Processing (NLP) and utilized the pre-trained model BERT to encode and decode and therefore generate images [2]. We also considered using transformers and self-attention models because they have been shown to have great success in NLP and maintaining relationships between information that are far apart (e.g. two pixels on opposite ends of the image). However, we decided not to implement this idea after consulting our TA, as self-attention was successful mainly due to the fact that it can relate important things that are far apart in larger images. This doesn't really apply to our project, as the image that we are analyzing is concentrated in the middle, and the letters are small enough such that self-attention wouldn't bring enough benefits to justify the computation costs.

Generative models, and GANs in particular, have also been used in font generation. For example, AGIS-Net attempted a task similar to ours, and achieves great results using a variation of GAN, but a key difference is that our network is learning both the shape of a character and the style, whereas AGIS-Net resembles that of a texture transfer as it feeds in both the styled image and an image of a "standard" character as well [3]. Style transfers in font generation have been explored extensively as a frequent approach to the problem of applying a certain style to some standard set of characters (Wang 2020). Our goal is more than just a style transfer; we want to be able to predict the resulting alphabet without using a baseline character as reference for the structure of a letter.

Another work in font generation similar to ours is GlyphGAN [4]. GlyphGAN is a true GAN in that it takes no specified input about the style of font to generate; rather, it takes a randomized vector as the input for the style, and then generates an entire font using the random style. GlyphGAN closely follows the architecture of DCGAN, with slight modifications. Our goals and our approach are slightly different from theirs. Instead of generating a font with a completely random style, we want to be able to generate the rest of a font given just a single letter of the font. In essence, we want to assist the designer, instead of replacing the designer. Therefore, instead of using a traditional generator like the one in DCGAN or GlyphGAN, we use a supervised autoencoder. A supervised approach makes sense in this case because we have access to complete fonts as ground truth data, and because we wish to generate fonts based on a given input, not just random fonts. We chose to use autoencoders because they have been shown to work well with image generation tasks, such as regenerating digits in MNIST [5]. The encoder portion of an autoencoder performs a process similar to principal component analysis, as it tries to distill an image down into a lower-dimensional representation. Then, the decoder portion will attempt to recover the original data from this lower-dimensional representation. The idea behind using autoencoders is that we will be able to encode the style of a font by looking at one character, then unwrap this encoding to generate images of every other character, using the same style.

3 Dataset

The dataset was sourced from Google Fonts, and it includes approximately 3900 fonts (including different styles of each font family) as .ttf files. Since .ttf files are difficult to work with in machine learning models, we had to convert everything to image files. To the best of our knowledge, there isn't any large dataset consisting of clean, well-styled images of fonts, so we had to make our own. To generate the input images for our model, we used the FreeType Python library to convert each letter in a font to an image. Color was not an important factor in the fonts, as we only care about the style and shape of the characters. Therefore, to save space and to reduce dimensionality, we chose to make all images grayscale, with 1 channel. To minimize the amount of blank background space around the characters in the image, we also center-cropped each image to a specified size. To begin, we simply iterated through every font downloaded from Google Fonts, loaded every character in that font in a python script, and then saved each character to its own file. To make the files convenient for saving and loading, we just saved them as .npy files, which is just the np array object, instead of .png files. We also started with just the uppercase alphabet, as we wanted to make sure the basics worked before extending to the rest of the font characters.

A problem that we ran into while creating our data set from the .ttf file is that some fonts tended to be a lot larger than others; that is, some styles of font are designed to be really tall or really wide, so even if we set the pixel size of the font to, say, 32 pixels, the height and the width of a particular character could still be more than the constraint. A number of fonts in the original set were too large, so we had to decide on a font size and image size that would fit most of the fonts but also retain a clear quality. After testing several combinations of font size and image size, we decided on a font size of 48 pixels and an image size of 64 by 64 pixels. After trying smaller font sizes and image sizes, we noticed that the model was having a lot of difficulty dealing with very thin fonts whose edges occupied only 2 or 3 pixels, so we wanted to increase the resolution so that it would be able to more easily recognize the edges of the thinner fonts.

With a larger font size, too many fonts which were very wide or very tall were getting cut out, and with a larger image size, the network would take too long to train. Additionally, there were several fonts which did not render correctly, and the FreeType library wasn't equipped to handle these cases (no exception or return values that would have indicated failure in loading a specific character), so we hand-removed these fonts (which took a very considerable amount of time and effort to go through almost 4000 fonts, and we did this two times). We also had to remove a couple of fonts that were so abstract that we weren't sure if they were loaded incorrectly or if they were supposed to look the way they did. After removing the remaining fonts that were too wide or too tall, we retained about 3800 fonts. The script used to clean up the fonts and generate the images can be found on our GitHub repo, along with all the resulting images.

All images were normalized from pixel values in [0, 1] to pixel values in [-1, 1] when we load them for the training process, since a mean at 0 is generally preferred. We also split the data set into a training

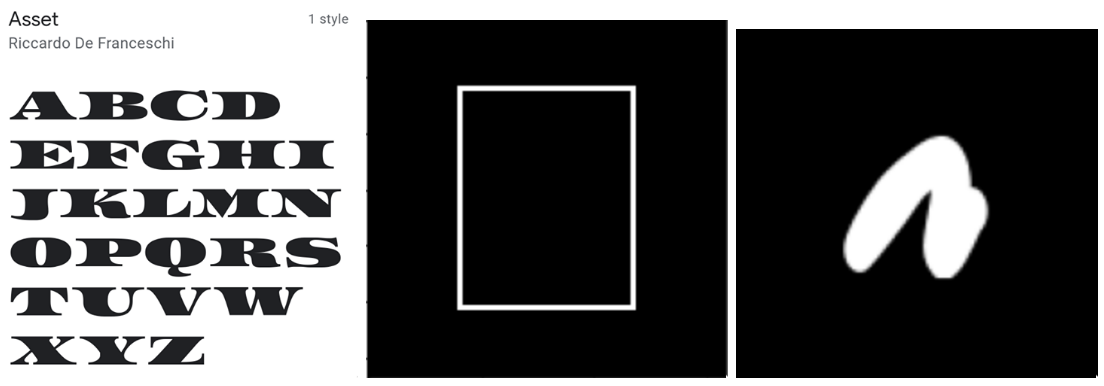

Figure 2: From left to right: (1) A font that is too wide to fit within the pixel limit, (2) A font that doesn't load properly, (3) A style so abstract that it's difficult even for humans to tell what it's supposed to be (it's actually a "b" - could you figure it out?).

set and a validation set which are 80% and 20% of the total data set, respectively.

4 Architecture

In this section we will discuss the different architectures that we attempted as well as the progressions we made. Our model was initially designed to take a single character of a font, and output one other character (instead of all characters) with the same font style as the given character. This allows us to assess preliminary results and make modifications to the network before training on every character. We found this method to be extremely valuable as we were able to save a considerable amount of time and computing usage. For our initial model, we chose to use the letter R to predict the letter B (i.e., predict what the letter B would look like in the given font style). The intuition behind this choice is that R has a lot of the features that are also found in other letters. Although we don't have empirical evidence to support this claim, R's generally have straight, vertical lines, slanted straight lines, rounded curves, and loops. The letter seems to have a lot of the features that are found in other letters, so we thought that R would be the best choice to use when predicting other letters. The choice of B however was completely arbitrary.

4.1 Autoencoder

The generation of characters was done with an autoencoder network. Because we had the actual character images for every letter for a given font, we were able to use a supervised learning approach here, by comparing the outputted image to the image of the target character.

We began with a very simple network: just a series of convolutions, pooling, and unpooling that increases the number of channels and reduces the image dimensions before decreasing the number of channels back down to one and changing the image dimensions back to their original sizes. Needless to say, it didn't work well at all, but one important discovery we made was that after training for a number of epochs, a grid-like structure as well as significant artifacts started showing up in our generated images. We attributed this to the use of pooling layers, because the unpooling layers would end up bringing white pixels to parts of the image where they didn't belong (i.e. the edges of the image), and unpooling makes it so that every $2x2$ section in the resulting image gets at least one white pixel.

To solve the problem of artifacts, we attempted a few solutions, one of which was using strided convolutions and strided transpose convolutions in place of the pooling and unpooling layers. The intuition behind this was that this allows the network to learn its own method for reducing image sizes instead of only doing max pooling, and hopefully also learn to keep the white pixels when they are close to the center, but not at the edges. However, artifacts persisted. Therefore, changing the method by which we were reducing and increasing image size didn't seem to be the solution. Also, we tried training the network for more epochs, and we saw that this helped the generated letter to have a more similar shape to the desired letter (Figure 3).

We also found that using ReLU as our activation function led to many artifacts in the background of our output images, which we suspected could be because ReLU wasn't zero-centered and because the gradients for negative inputs were just zero. To try and mitigate this issue, we instead used LeakyReLU, so that it would be closer to being zero-centered and the gradients would be nonzero. In our initial

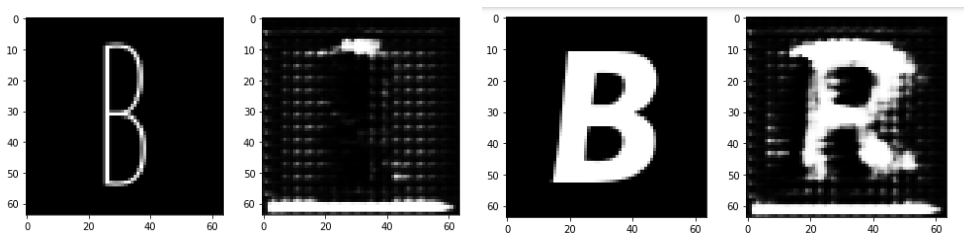

Figure 3: Two pairs of images are shown (each pair of images has ground truth on the left and the generated image on the right). Left pair: the grid problem in the first network; the model did not predict the letter well and still had the grid and a significant amount of artifacts at the bottom. Right pair: strided convolutions did not fix the grid problem, but it appears possible to predict the letter, although the model hasn't learned the shape of the desired letter yet.

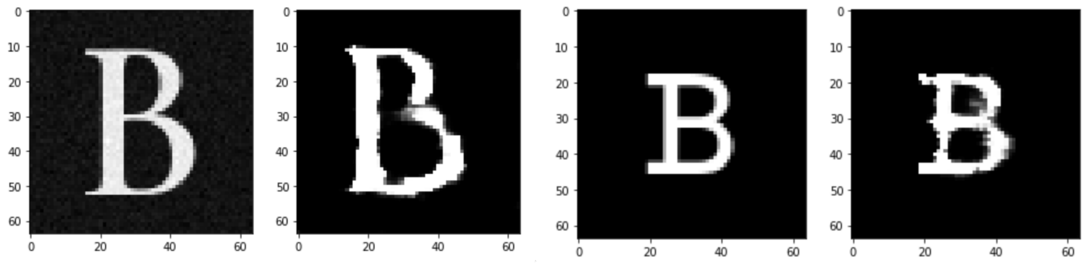

Figure 4: Examples of jagged edges and speckling throughout the letter

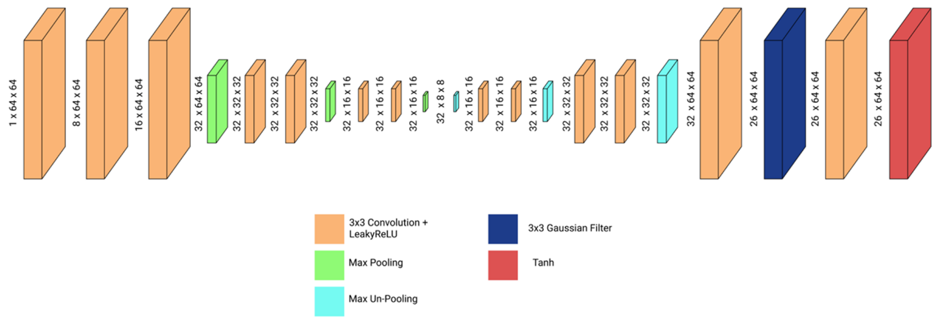

Figure 5: Architecture used in the autoencoder

testing, we also noticed that there were a lot of jagged edges and speckling within the output images. We tried adding a Gaussian filter at the end of the network, but this only ended up blurring the image. Therefore, we decided to move the Gaussian filter further into the network, by applying the Gaussian filter, then 3 additional convolutional layers, and then the final activation function. Because the images are normalized to [-1, 1], we use tanh as the final activation function to compress all values into this range.

We theorized that the best way to extract the style of the letter would be first reducing the size of the image down to just the letter first, because most images have a black border surrounding the actual letter itself. Then, we can extract the features more easily through traditional convolution layers in the encoder part before decoding the feature/letter representation back to the original image size. The first part appears similar to a cropping object problem and we found that pooling and convolution at the smallest image size have been found to be successful at cropping images tasks [6]. Therefore we modified the architecture accordingly: we switched back to pooling/unpooling and then added more convolution layers in between pooling.

The resulting autoencoder architecture consists of convolution layers, LeakyReLU layers, max pooling and unpooling layers, and a Gaussian filter layer, all ending with a Tanh layer. The architecture is shown in Figure 5.

After training and testing with the architecture that generates the letter B from the letter R, we worked on an architecture that extends from the letter B to all of the alphabet. To accomplish this, we thought of two possible methods: (1) we can just duplicate the same architecture, and run it 26 times, one for each letter in the alphabet or (2) we can modify the current architecture and change it so that the autoencoder generates 26 images, each in its own channel, at the end of the network. We decided that the first way isn't great, because having to repeat the same thing for every character seemed inefficient—we will be encoding the style of the letter every single time when the style representation could probably be used across the alphabet. Therefore, we chose the second route, hoping to get a good style representation of the font through the encoder portion and then when we are decoding, we do a series of convolutions that eventually ends up with 26 channels, one for each letter in the alphabet.

4.2 Discriminator

When using only the autoencoder, the network was doing well at recognizing the style of the fonts, but it was having trouble understanding the general shape of the character it was predicting. The

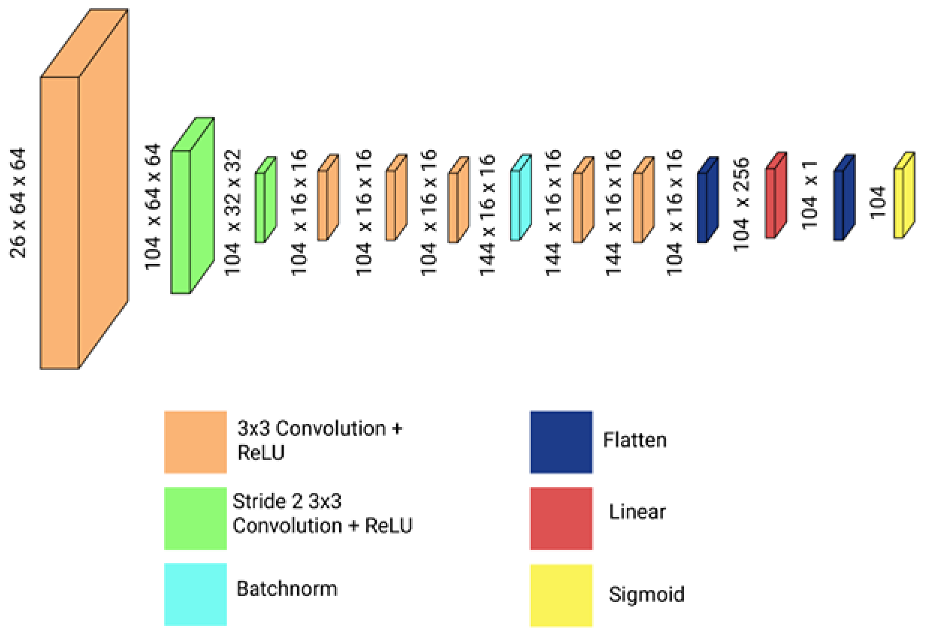

Figure 6: Architecture used in the discriminator

discriminator network was designed to distinguish between real and fake images of the target letters, so that the autoencoder would have a better baseline understanding of what the target character looked like.

The discriminator is a simple classifier, using convolution layers and max-pooling layers to generate a prediction for each font (a set of 26 characters): whether the given image for each character is a real character or a generated character. The architecture of this network closely follows the architecture of the discriminator in the DCGAN paper. In other words, the job of the discriminator is to learn to differentiate the generated fonts and the real font, and to help it learn, we label all the original images as the real fonts while labeling any generated images as fake when inputting them into the discriminator.

5 Training

The network was trained in 3 steps:

1. 200 epochs with a learning rate of $2 \cdot 10^{-3}$ and a batch size of 16
2. 40 epochs with a learning rate of $9 \cdot 10^{-4}$ and a batch size of 16
3. 40 epochs with a learning rate of $6 \cdot 10^{-4}$ and a batch size of 16

For both the autoencoder and the discriminator, we used the Adam optimizer with parameters beta1 $= 0.9$ and beta2 $= 0.999$. Each batch consists of 16 fonts, and for each font, we take the 26 uppercase letters in that font for the batch.

To train the network, we alternately trained the autoencoder and the discriminator. The discriminator has to learn faster than the autoencoder, since the autoencoder will not learn anything from the discriminator if it is always able to trick the discriminator. Therefore, within each iteration of our training loop, we updated the discriminator twice and the autoencoder once. We also experimented with the number of extra iterations to train the discriminator vs the autoencoder, but given the limited computing power we had, even training one extra time took a significant amount of time. Luckily, we found that it was sufficient, and the autoencoder was able to learn from the discriminator throughout the entire training process.

To evaluate the discriminator, we simply use the cross entropy loss after inputting real font images labeled as real, and generated font images labeled as fake. We compute the backward gradients and update the weights accordingly after each of the two batches.

To evaluate the autoencoder, we use a combination of 4 losses. First, we compute the L1 loss over the entire image. Then, we compute the multi-scale structural similarity (MS-SSIM) between the generated image and ground truth image of the target font, and subtract this value from 1 for our second loss. Finally, to ensure smoother local structure, we compute the local L1 loss over 3 random 16 pixels by 16 pixels patches within the images. Finally, we have the cross entropy loss from the discriminator, when we feed it our generated images labeled as real images. These losses are weighted according to the suggested L1/SSIM weighting in "Loss Functions for Neural Networks for Image Processing" (Zhao et al. 2015), and also by their relative magnitude:

$$\mathcal{L} = 9 \cdot 10^{-3} \cdot \mathcal{L}_{disc} + 0.16 \cdot \mathcal{L}_{L1} + 0.84 \cdot \mathcal{L}_{SSIM} + \mathcal{L}_{local}$$

5.1 Loss Functions

5.1.1 L1 Loss

We use a classic L1 loss function, defined below, as one of the loss functions to train our autoencoder.

$$\mathcal{L}(x, y) = \frac{1}{N} \sum_{i=1}^{N} (l_i), l_i = |x_i - y_i|$$

The L1 loss function does well in preserving colors and luminance, but fails to understand local structure. To try and improve understanding of local structure, in addition to the L1 loss calculated over the entire image, we also calculate the L1 loss over 3 randomly chosen 16 by 16 patches in the image.

5.1.2 Structural Similarity Loss

The structural similarity (SSIM) between two windows x and y has 3 components: luminance $l(x, y)$, contrast $c(x, y)$, and structure $s(x, y)$. They are given by the formulas below (note: each C_i term is a constant): [9].

$$\begin{aligned} l(x,y) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \\ c(x,y) &= \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \\ s(x,y) &= \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \end{aligned}$$

Setting $C_3 = C_2/2$ (which is necessary for calculating SSIM from the above 3 values), we have

$$\begin{aligned} \text{SSIM}(x,y) &= l(x,y) \cdot c(x,y) \cdot s(x,y) \\ &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \cdot \frac{2\sigma_{xy} + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \end{aligned}$$

Then, we can define a structural similarity loss function [10] as

$$\mathcal{L}(x,y) = 1 - \text{SSIM}(x,y)$$

The means and standard deviations are computed with a Gaussian filter with a specified standard deviation σ_G. Instead of using the standard implementation of SSIM with a set Gaussian filter size, we instead use a variant called multi-scale structural similarity (MS-SSIM), which computes the structural similarity across several different filter sizes.

We decided to include this loss because it takes into account the values of neighboring pixels, unlike L1 loss, which only looks at the difference between two corresponding pixels. However, it still does have some downsides. For one, SSIM is not very sensitive to a uniform bias across the entire image. This could cause issues when the predicted image is uniformly duller or brighter than the ground truth image, which the SSIM loss would have a hard time detecting [10].

6 Results

To test our model, we generated every uppercase letter in a font by inputting a single character R from the font. We chose the letter R based off of its properties of having vertical/horizontal edges, slanted edges, and curved edges. All of the fonts used to test/evaluate our model are from the validation set, which the model had never seen before during training. As a quick reminder, we initially divided our total font dataset into 80% training and 20% validation.

The performance of our network is most naturally assessed qualitatively by visually comparing the outputted images to the ground truth images. Our network was able to pretty accurately replicate the style and structure of many fonts, but still had some defects. One issue was that the generated fonts were not particularly sharp, and often lacked clean edges, in comparison to the ground truth fonts. In addition, the network had trouble predicting thinner fonts and fonts with less standard styles. We also noticed that letters more similar to R, such as B, D, or P, achieved the best results, while less similar letters, like A or G, looked noticeably worse in terms of sharpness and structural similarity.

To quantitatively assess the performance of our model, we decided to compute the average L1 loss and the average SSIM loss for each letter. This allows us to analyze which letters were easier to generate from the letter R and which letters were more difficult to generate.

The loss for each letter shows some pretty interesting results. The letter R had the lowest L1 Loss and SSIM loss across the alphabet, as expected. The letters I and T also had relatively low L1 loss and SSIM loss. The letter W had the highest L1 loss and SSIM, followed by the letters M and Q. Note, however, that the letters that seemed to do the worst based on numerical loss alone were not necessarily the same letters that seemed to look the worst by eye. For instance, as seen in figure 10, W, M, and Q actually had very reasonable generated images—the styles were visually accurate even though the losses were high. We think that this is due to the fact that these letters have more lines/have more pixels in the letter itself, and when the white pixels cover more area, it's also easier to have high losses even though most of the pixels are correct. In other words, even if the same percentage of the white pixels are generated correctly, the losses would still be higher.

Letter	A	B	C	D	E	F	G	H	I	J
L1 Loss	0.0767	0.0577	0.0600	0.0607	0.0523	0.0459	0.0798	0.0650	0.0314	0.0809
SSIM Loss	0.0982	0.0633	0.0869	0.0700	0.0740	0.0693	0.1035	0.0823	0.0617	0.1548

Letter	K	L	M	N	O	P	Q	R	S	T
L1 Loss	0.0663	0.0408	0.1399	0.0790	0.0686	0.0470	0.1246	0.0307	0.0619	0.0410
SSIM Loss	0.0795	0.0666	0.1799	0.0928	0.0790	0.0555	0.1638	0.0240	0.0843	0.0660

Letter	U	V	W	X	Y	Z
L1 Loss	0.0652	0.0623	0.1467	0.0599	0.0560	0.0504
SSIM Loss	0.0858	0.0835	0.1933	0.0735	0.0826	0.0692

Table 1: Average L1 and SSIM loss for each character of the alphabet.

The loss plots for the training process can be found in Figures 7-9. These plots show the changes in each of the losses for each batch used in our training process (note that we recorded the losses once every 50 batches). As expected, the losses decreased as training progressed and eventually leveled off.

We also saved the generated fonts as pictures so we could get a better idea of how they actually look (see Figure 10). From first glance, the generated fonts look pretty good. Although they do not look exactly the same as the ground truth fonts upon close inspection, the resemblance is obvious. In Figure 10, we show three fonts with rather distinct styles: one font that is relatively thick, another that is relatively thin, and one from the Serif family (marked by the extra lines on letters like I, J, T, etc.). More detailed qualitative analysis can be found in the discussion section.

7 Discussion

Overall, our network performed reasonably well, and was able to predict the general style and structure of fonts based off of just a single letter. However, there were still many areas in which the network was lacking. We suspect that many of the issues present in the final results were due to limited data

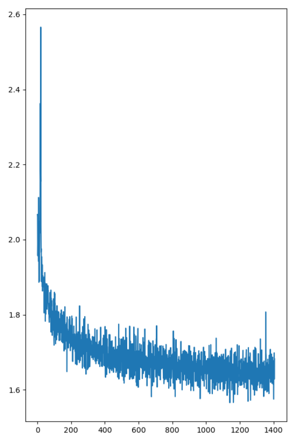

Figure 7: L1 Loss of the model throughout training

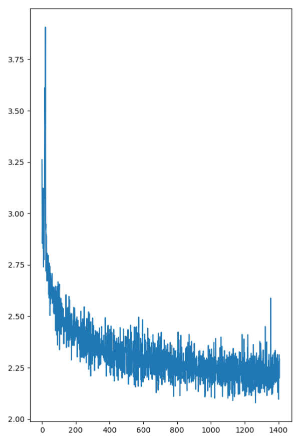

Figure 8: SSIM Loss of our model throughout training

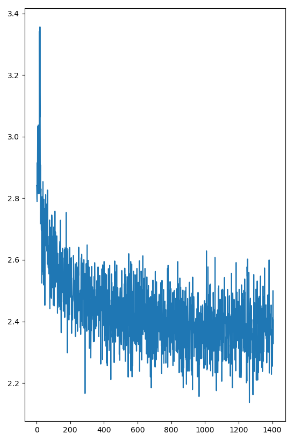

Figure 9: Local L1 loss of the model throughout training

and limited computing power, but we also have some ideas as to how we could improve this model to account for the problems that we ran into.

One of the biggest issues was the uniformity of our data. Many of the fonts we pulled from the database had similar styles and structures, with many of them being very similar to a classic font like Arial. This meant that when we encountered something like a cursive font or a "blocky" font, our network wasn't able to generalize to predicting the letters of these fonts well (see Figure 11). With a larger dataset including a larger variety of fonts, we suspect that the model would have been able to generalize to these fonts better. Another way that we could better learn different styles might be using more than just the letter R to learn the style. Providing more letters as input into the autoencoder would probably make letters much easier to generate.

Another issue we had which we weren't sure how to address was the variation in structure within the same letter. For example, most fonts contained the regular 'A' character, but some fonts had the cursive 'A' character. These are two valid structures of the character 'A', but they don't really have much in common. We suspect this caused some trouble when training our discriminator network, since it wasn't able to learn the much less common cursive 'A' structure. This caused our model to output a normal-looking A even when the target 'A' was cursive. Furthermore, this phenomenon isn't limited to just cursive characters; the problem extends to a lot of the fonts with untraditional shapes, and it was difficult for our model to generate them, mainly due to the lack of variations in our dataset.

In addition, many fonts with thin lines proved difficult for our network to predict. Since these thin lines were often only 2 or 3 pixels wide (and in some cases only 1 pixel wide), the network had a hard

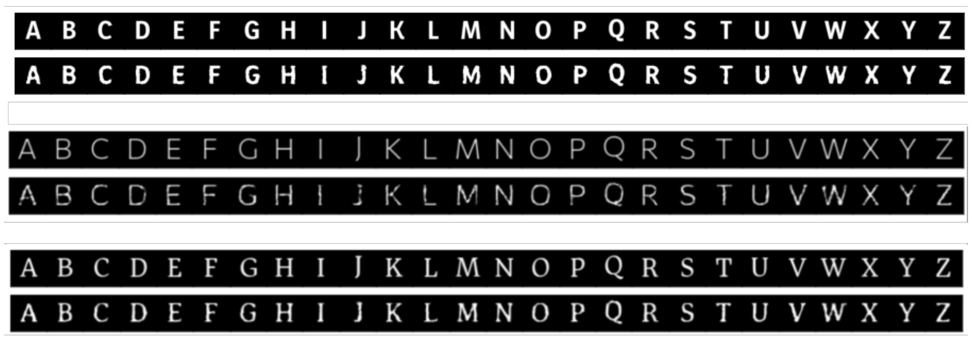

Figure 10: Examples of three sets of complete fonts generated from only the letter R. In each set, ground truth (i.e. the original font) is shown in the top row, while the generated font is shown in the bottom row.

time differentiating these lines from the background (see figure 12). We ran into this issue when training with smaller images at a smaller font size, so we decided to scale up the resolution to our current image size and font size. Although it helped a little, it seems that it wasn't enough to completely solve the problem. With a limited amount of computing power, we didn't want to scale up the image any further, as this would make the network much slower and much harder to train. However, we still believe that higher resolution images could have solved this issue.

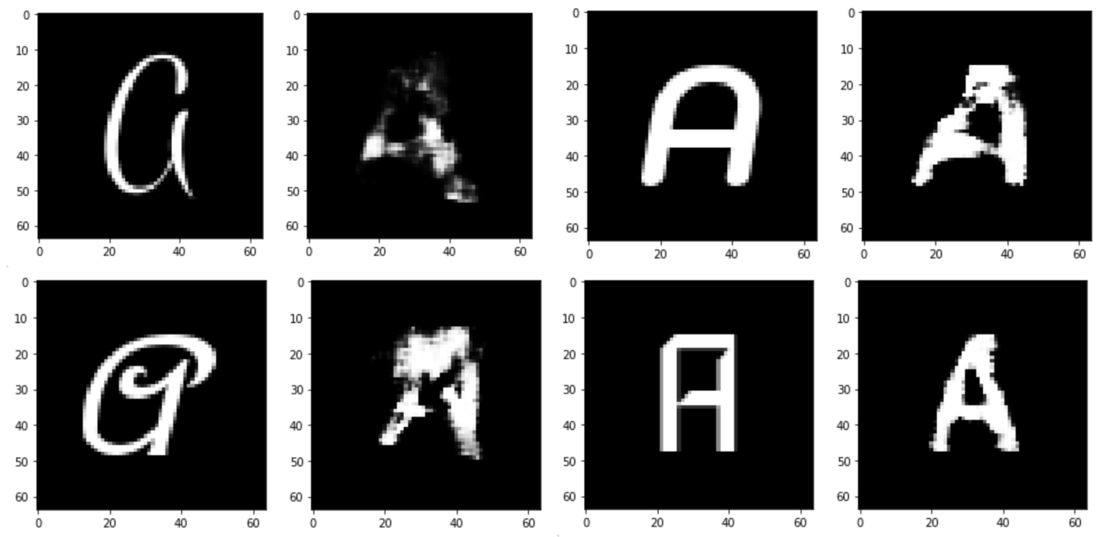

Figure 11: Examples of variations in structure that our model has difficulty generating correctly.

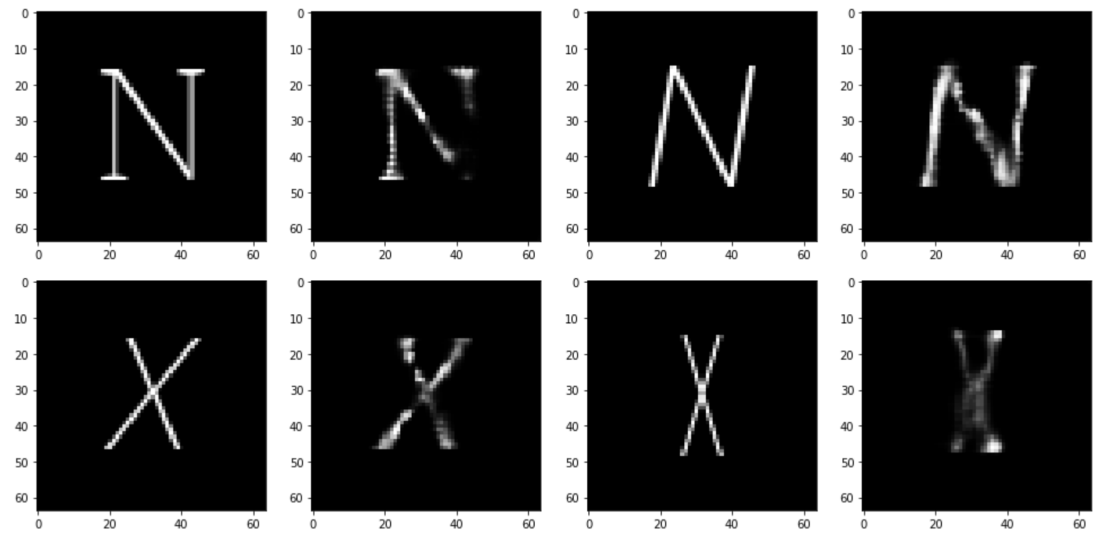

Figure 12: Examples of thin fonts that our model has trouble generating.

Another problem with the fonts is that sometimes the pixels of the letter in the images aren't completely white; the letter can be partially gray, which makes it even harder for our network to differentiate between the foreground and the black background. This appeared to be a prominent issue for fonts with thin lines, which makes the matter even more difficult as discussed previously. An issue that this led to is that in the generated image, the letter is just generally darker and has lower pixel values than the ground truth. A possible solution to this problem might have been doing more preprocessing on the image, such as simply changing all pixel values above a certain threshold to 255 to ensure that all parts of the letter are white or increasing all pixel values above a certain threshold by some constant. However, this may cause an issue for fonts with thicker width and fonts with more fade off between the letter and the background. Perhaps this issue could also be resolved by finding better ways to convert from .ttf to .png, but we didn't have time or the computing power to explore this aspect.

Also, we noticed from the final results that our quantitative results, the L1 and SSIM loss, didn't exactly correspond to our qualitative results, the generated image. For example, although the letters W and M didn't achieve particularly great losses, their style and structure in general seem reasonably close to the ground truth. We suspect that this may have been due to size differences in the different letters, as W and M are generally larger and occupy more pixels than other letters. Therefore, tuning or altering our loss function to control for the size could have helped us achieve better results.

We've talked about many different ways in which our model is lacking, but one of its biggest successes is accurately learning the style of the font from just one letter. Even without being able to ascertain the structure of a certain letter, we could see that the style was being replicated to a good degree. For example, you can see an obvious slant in the the generated images for slanted fonts, and you can see the line width vary from thin to thick based on the style of the given character (see figure 13).

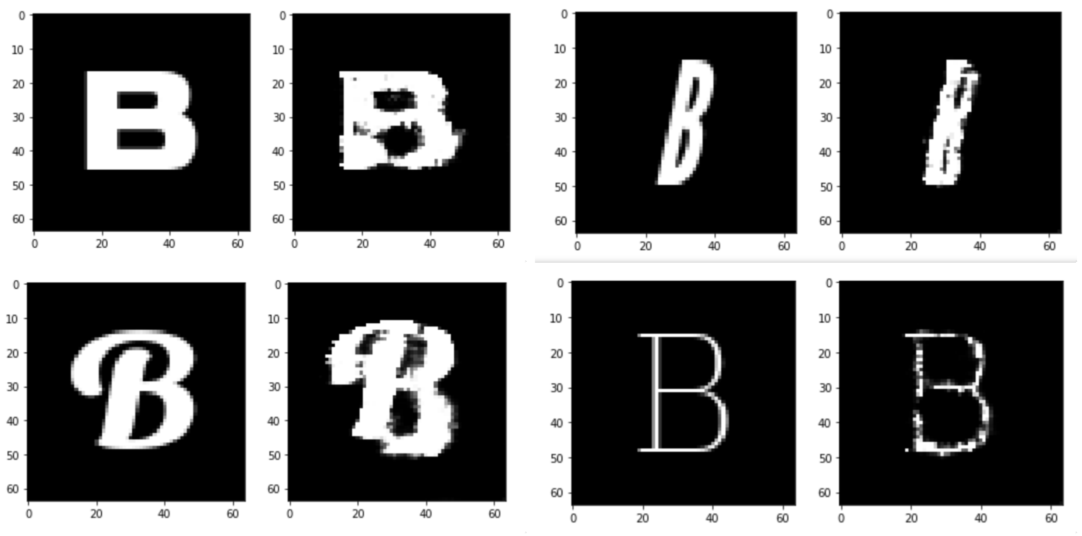

Figure 13: Some examples from our earliest iterations of training, where it was obvious that even when our letter didn't have great structure, it was apparent that it was learning the style of the font.

It should be noted that fonts are almost always in the .ttf format, which means that in order to actually use our generated font images, we need to first convert them into .ttf files. Thus, we would first save the generated images (numpy arrays) as .png files, and then convert them into .ttf files. We did not implement this step, but we feel that it is a logical next step, after improving the model to generate sharper, smoother images.

8 Acknowledgements

We would like to thank our professors, Professor Russakovsky and Professor Deng, for teaching the course and providing us with the background to take on this project. We would also like to thank our advisor Jiaqi for answering our questions and providing valuable advice throughout the process.

References

[1] Mark Chen et al. "Generative Pretraining from Pixels". In: (2020).

[2] Jacob Devlin et al. In: *Proceedings of the 2019 Conference of the North* (2019). DOI: 10.18653/v1/n19-1423. URL: http://dx.doi.org/10.18653/v1/N19-1423.

[3] Yue Gao et al. "Artistic glyph image synthesis via one-stage few-shot learning". In: *ACM Transactions on Graphics* 38.6 (Nov. 2019), pp. 1–12. ISSN: 1557-7368. DOI: 10.1145/3355089.3356574. URL: http://dx.doi.org/10.1145/3355089.3356574.

[4] Hideaki Hayashi, Kohtaro Abe, and Seiichi Uchida. *GlyphGAN: Style-Consistent Font Generation Based on Generative Adversarial Networks*. 2019. arXiv: 1905.12502 [cs.CV].

[5] G. E. Hinton and R. R. Salakhutdinov. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), pp. 504–507. ISSN: 0036-8075. DOI: 10.1126/science.1127647. URL: https://science.sciencemag.org/content/313/5786/504.

[6] Peng Lu et al. *An End-to-End Neural Network for Image Cropping by Learning Composition from Aesthetic Photos*. 2019. arXiv: 1907.01432 [cs.CV].

[7] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. arXiv: 1511.06434 [cs.LG].

[8] Mathew Salvaris, Danielle Dean, and Wee Hyong Tok. “Generative Adversarial Networks”. In: *Deep Learning with Azure* (2018), pp. 187–208. DOI: 10.1007/978-1-4842-3679-6_8. URL: http://dx.doi.org/10.1007/978-1-4842-3679-6_8.

[9] Z. Wang, E. P. Simoncelli, and A. C. Bovik. “Multiscale structural similarity for image quality assessment”. In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*. Vol. 2. 2003, 1398–1402 Vol.2. DOI: 10.1109/ACSSC.2003.1292216.

[10] Hang Zhao et al. “Loss Functions for Neural Networks for Image Processing”. In: *CoRR* abs/1511.08861 (2015). arXiv: 1511.08861. URL: http://arxiv.org/abs/1511.08861.