Generative Typefaces

Exploring the latent world of named fonts
to help create new ones

Jerry Roback
Springboard Jan ’21 Cohort
July 24, 2021

Contents

Background

Like fashion and architecture, typefaces (or fonts) can give us a glimpse into culture at a specific moment in time. A particular typeface can be used to communicate humor, formality, simplicity or a modern and trendy point-of-view. The shape and style of each letter varies subtly, lending to unlimited possibility. With written word all around us on signage, in print, online, and beyond—type design has become an important part of modern culture.

According to Google, there have been over 130,000 typefaces published for the Roman alphabet. Each, at a minimum, consists of 26 characters, 10 digits and a dozen or so punctuation marks. Creating custom typefaces is an activity commonly done for brands in the marketing world, with many desiring something unique to their brand or corporate identity. One major reason that brand design projects can garner 7-figure price tags is the development of a custom font.[1] The typeface design process can quickly consume hundreds of creative hours and many rounds of iterations making it an expensive activity to undergo using traditional methods.

[1] Business Insider, https://www.businessinsider.com/heres-how-much-the-worlds-most-iconic-logos-cost-companies-2013-3

Goal & Objectives

Goals:

a. Can machine learning make the design of new/custom typefaces more accessible?
b. Can a new-to-the-world typeface be 'designed' using generative methods?

Project objectives:

a. Enhance understanding of the typography landscape
b. Empower designers and creatives with a more intuitive way to explore type
c. Create new efficiencies around the type design process

Data and Scope

Fonts were collected from two sources: Google Fonts and Landor & Fitch. Google Fonts is a robust catalog of open-source fonts that's striving to make it as easy as possible to integrate expressive type design on the web.[2] The project maintains an open repository on Github that includes over 1,100 fonts. In addition to these publicly available fonts, my client and employer, Landor & Fitch, provides licensed access to over 2,500 fonts through a private cloud. Both Google Fonts and the font cloud provide fonts in the form of TrueType Font (.ttf) files. The .ttf file format contains mathematical representations of each character within a named and published font.

Unfortunately, no official dataset typeface or Roman character dataset existed prior to this work. A large part of this project included the construction of an image dataset from the available 3,500+ .ttf files. Using image processing methods, I created uniform images out of the mathematical representations of each character, so that each image in the dataset represents a single, 224x224px grayscale image of a typeface character. Because of the variance in special characters and discretionary characters included between fonts, the scope was limited to 62 characters in the Roman alphabet—uppercase and lowercase letters (52) as well as digits (10). Due to computing constraints, the scope was refocused to two select

[2] Google Fonts https://fonts.google.com/about

characters in the dataset. Final analysis was conducted on lowercase ‘a’ and ‘j’ characters only.

To clean this custom dataset, a Convolutional Neural Network was trained to help classify and filter out unreadable characters which lead to less noise in each class. An Autoencoder was trained to help visualize the latent space and two Generative Adversarial Networks (GAN) were trained to generate new characters—one unconditional and one to generate conditionally on the character class.

Summary of Results

A conditional Deep Convolutional GAN (DCGAN) produced the most convincing results. The DCGAN was trained on 7254 images for 50 epochs with the goal of generating ‘a’ and ‘j’ character pairs from random noise. A pair can be thought of as the starting point of a typeface. The two characters should look like they belong in the same family of typeface by demonstrating similar features. For example, slender ‘a’s with slender ‘j’s or thick and curvy ‘a’s with thick and curvy ‘j’s.

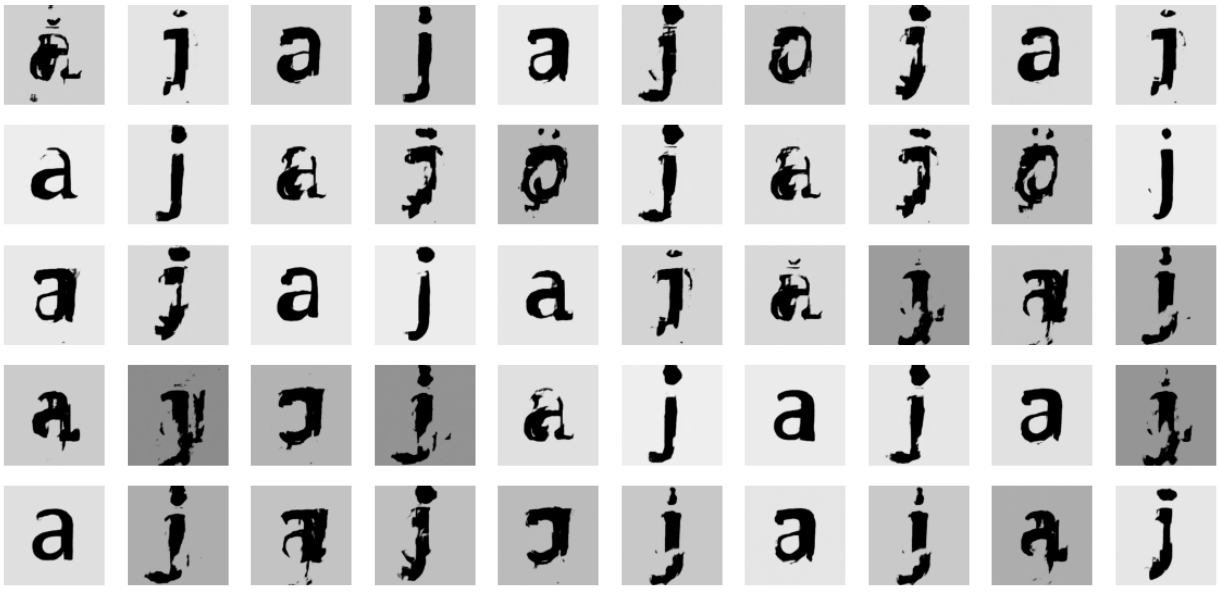

Figure 1. Output from DCGAN trained on 7254 images for 50 epochs (‘a’ & ‘j’ pairs), optimized

The 50 epoch DCGAN, as seen in Figure 1, was trained to address a variance issue in the characters coming out of the initial DCGAN. While the 50 epoch model could

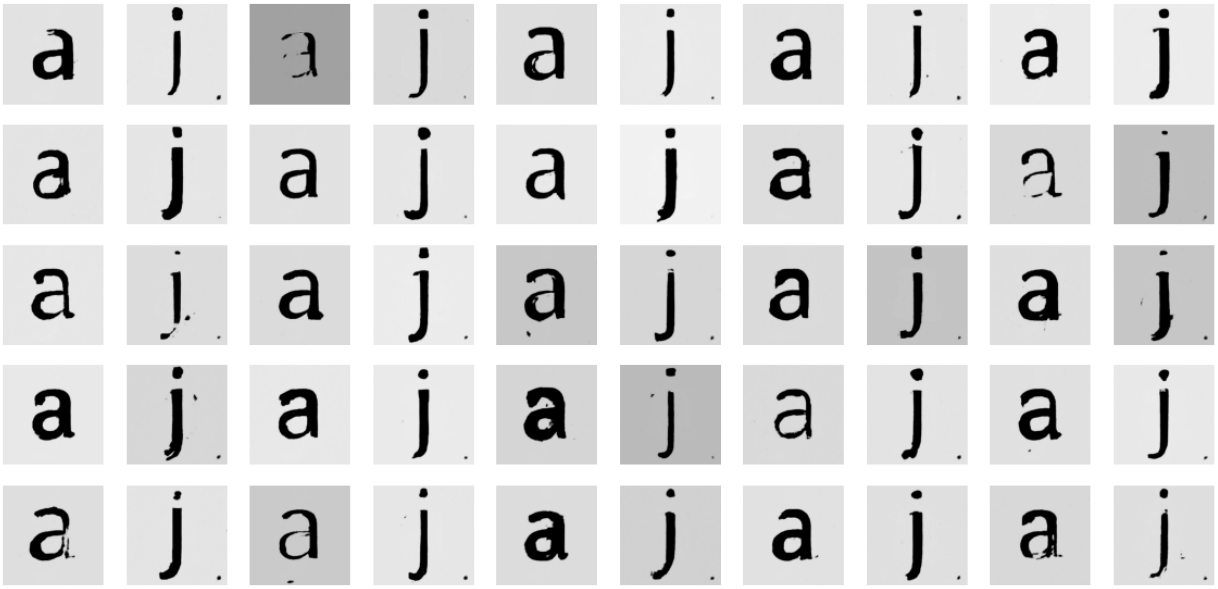

Figure 2. Output from DCGAN trained on 7254 images for 300 epochs (‘a’ & ‘j’ pairs), baseline

have benefited from additional training, the variance in the style of the ‘a' characters is much more apparent in the way they connect (rounded and fully connected at the top) and well as where they finish (straight tails, no tails and curved tails) than in the 300 epoch DCGAN as seen in Figure 2.

While the characters in the 300 epoch output are much clearer and sharper, all of the characters share the same shape and style. Most of the variation is observed in the thickness of the character which is not representative of the shape diversity found in the Exploratory Data Analysis phase.

Exploratory Data Analysis (EDA)

The goal of this EDA was to find trends and patterns. It was also used to construct, clean and shape the data in preparation for pre-processing. The first step was to use the computer's filesystem to construct an image dataset of characters. Each image in the dataset is a representation of a single typeface character and was constructed using 62 character representations from 3865 fonts (ttf files). The resulting dataset contains:

- 239,630 224px x 224px grayscale images (.png files) belonging to 62 classes
- A balanced set with 3865 images per class
- Each class represents a single character limited to uppercase and lowercase Latin characters (52) as well as digits (10)

After constructing the dataset from the filesystem, the Keras library was used to convert the image files into pixel matrices. When this data was visually inspected, a few errors were observed. The first observation was the presence of blank characters (all black pixel images) for characters that may not be present in specific typefaces. And the second was the presence of unreadable (white rectangular images) characters not representable by using a Latin keyboard. This identified some Arabic or other non-Latin alphabet fonts to be dropped from the dataset (Figure 3).

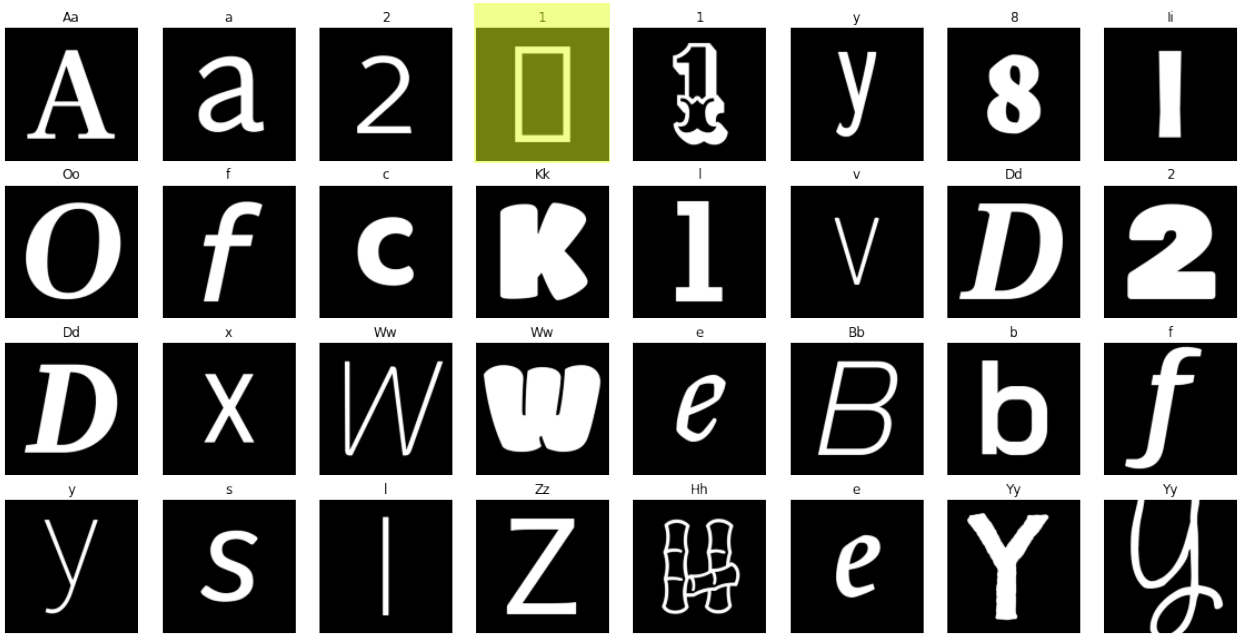

Figure 3. Output from dataset batch. Unreadable character example shown in yellow

The first approach at cleaning these characters from the dataset was to filter based on their image hash representations. While this approach worked quickly to remove an average of 170 blank and unreadable characters from each class, there were still a significant number of unreadable characters present in each class (the) that would affect the data (Figure 4).

Figure 4. Representative unreadable characters remaining in a class after hash filtering

The unreadable characters that remained after filtering showed a variance that was not observed prior to the filtering. The characters were not all identical, but rather rectangles and squares of different thicknesses, widths and heights. Rather than manually pruning through 225,000 images, an alternative, machine learning method was explored. Due to the consistency in shape (rectangular) and features (sharp corners, straight horizontal and vertical lines) this was a great task for an image classifier—or more specifically, a convolutional neural network (CNN).

Convolutional Neural Network (CNN) Image Filtering

An image classifier was trained to help increase the identification and filtration of unreadable characters. The intent of the classifier is to identify both unreadable characters and readable characters. For that reason, the model was trained as a two-class categorical classifier and not a binary image classifier. The CNN-based classifier was trained on a subset of 468 images from the original dataset balanced into classes of 'unreadable' images and ‘readable’ images separated into Training (70%) and Validation (30%) datasets. Unreadable images were transformed before training, including horizontal and vertical flip and a zoom range of 30% to help with generalization. The architecture of the CNN classifier used best-practices for image identification, including:

- Two convolutional layers with increasing size
- Followed by pooling and dropout layers to reduce the dimensionality
- Then, after flattening, a dense layer before the final activation layer
- The model uses the softmax activation function to predict a class of either 'readable' or 'unreadable' for each image

The classifier was fit with 98.9% accuracy on the validation set with a loss of 7.3%. This classifier was implemented to update the character filtering function with nearly as strong results. A visual inspection of the results on filtering for both the ‘a’ and ‘j’ classes showed that the classifier performed as expected. It filtered out 96% of the unreadable characters from each character class while erring on the side of over-filtering (false positives) versus under-filtering (false negatives) (Figures 5 & 6).

Character Filtering Image Classifier Confusion Matrix

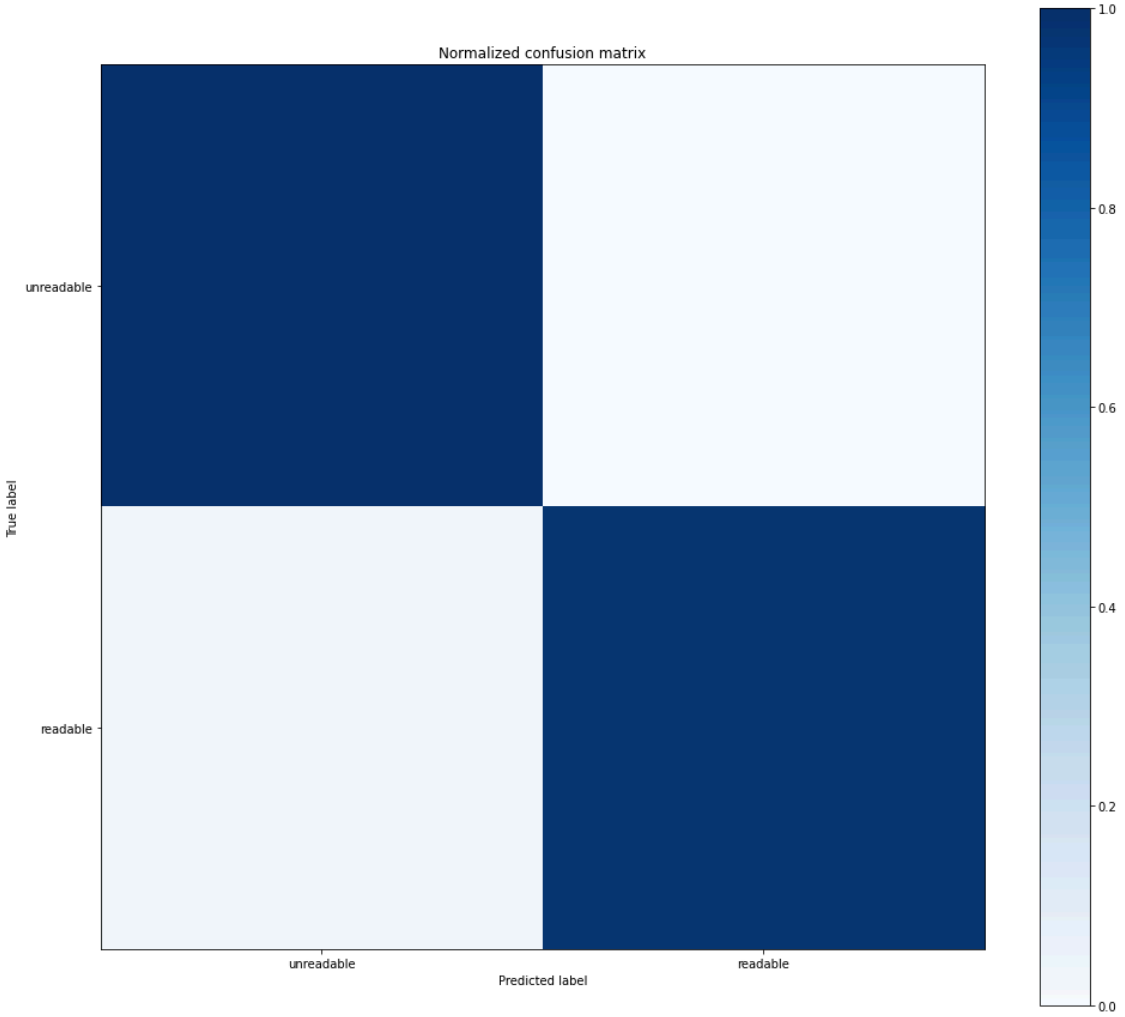

Figure 5. Confusion Matrix from the Character Filtering CNN

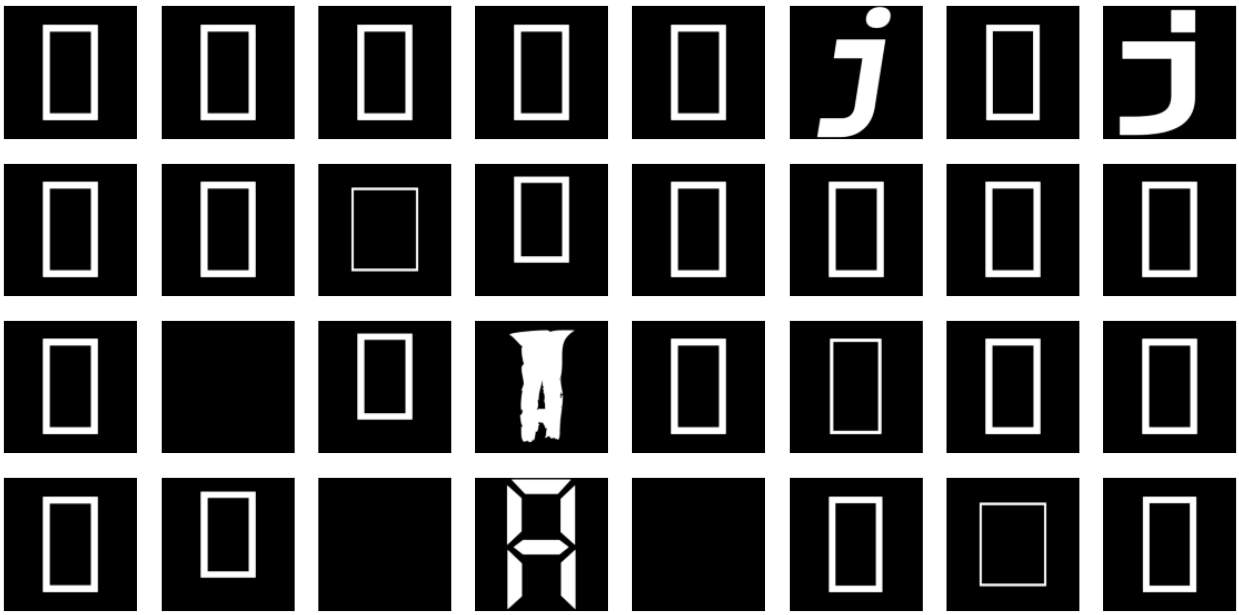

Figure 6. Representative Output: ‘a’ and ‘j’ characters classified as ‘unreadable’ by the CNN

From the output of the CNN, it was observed that many of the false positives shared features with the general unreadable character (Figure 6). Straight, wide horizontal lines appeared in the ‘a’s above, while the ‘j’s had strong vertical and horizontal lines. One ‘j’ showing a perfect square dot above the character.

Visualizing the Latent Space

With the ‘a’ and ‘j’ classes appropriately cleaned and filtered, the next step was to get an understanding of the latent spaces that made up each character. An autoencoder was constructed to help visualize the latent space and project each character along 2 dimensions. The autoencoder was trained to include 4 hidden layers of decreasing dimension and had a Mean Squared Error (MSE) loss of about 3.5% after 10 epochs (Figure 7).

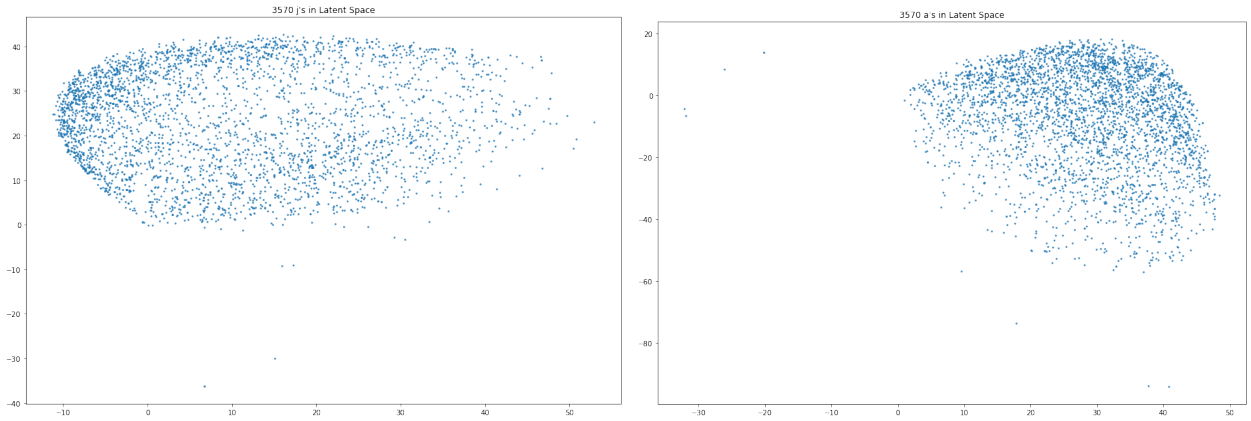

Figure 7. 3570 ‘j’s (left) and ‘a’s (right) projected into latent space

displayed the average/predicted character representations from specific zones using two characters as examples: 'j' and 'a'.

Using the 2 dimensional Autoencoder, Figure 7 showed all remaining ‘a’s and ‘j’s projected into their latent space representations. Figures 8 and 9 overlay about 10% of each class’s images onto the latent space to help visualize what the model learned about the representation of each.

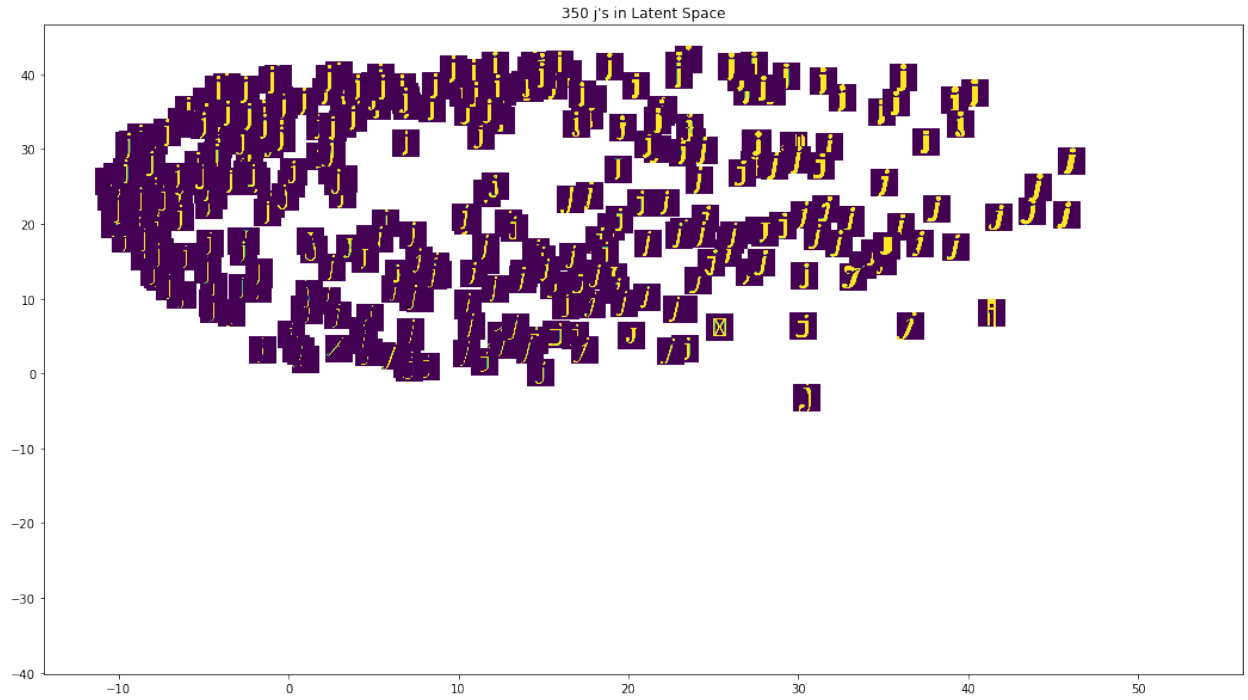

Figure 8. 350 ‘j’s projected into 2 dimensional latent space

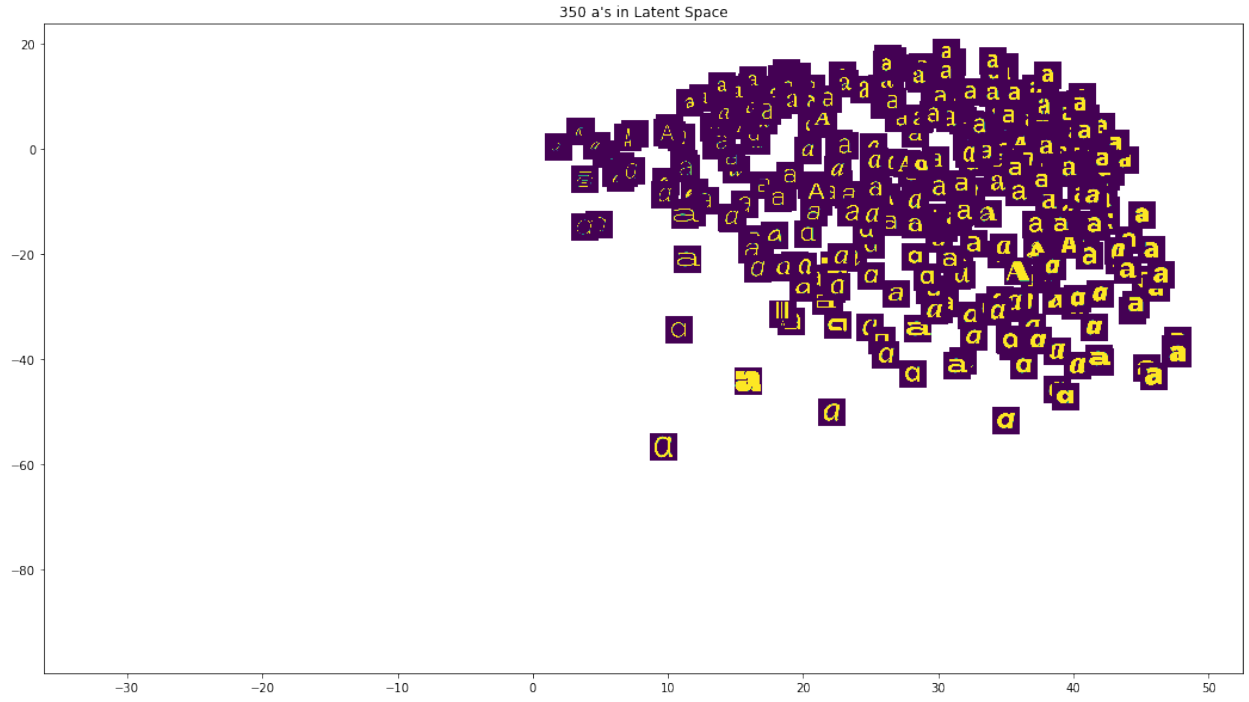

Figure 9. 350 ‘a’s projected into 2 dimensional latent space

Observations from the latent space visualizations showed that the thin ‘j’s were clustered on the left side of the plot, the thicker/bolder ‘j’s on the right, and the italic ‘j’s clustered toward the bottom of the distribution. The 'a' characters showed a similar distribution. The thinner 'a's appear on the left while the thicker 'a's appear on the lower right.

Exploring Latent Vectors

With the latent space of the characters defined, it was then possible to explore the latent vectors—interpolating between characters moving from left to upper right to both verify and help further visualize (Figure 10).

Figure 10. ‘J’ and ‘a’ latent space vectors (from codings) projected into 2 dimensional latent space

The output from the latent vectors (Figure 10) do verify what was observed in the overall latent space plots (Figures 8 and 9). The thinner characters occupy the left and the characters tend to get thicker and more italic the further we navigate to the right. More observable in the ‘j’s than the ‘a’s as the ‘a’s are clustered a lot closer together.

Generative Modeling

With the characters ready for processing and with a deeper understanding of the latent space, I explored multiple generative models in an attempt to recreate convincing ‘a’ and ‘j’ characters that appear as though they could be from the dataset. I trained two Generative Adversarial Network (GAN) architectures including a Deep Convolutional GAN (DCGAN) and a Conditional GAN in order to generate new and realistic typeface characters for a specific class.

The baseline DCGAN was trained using both Keras and TensorFlow libraries. The DCGAN was defined using best practices for image generation which includes use of the sigmoid activation in the discriminator, tanh activation in the generator, and the use of a leaky relu between convolutional layers to upsample and downsample (Figure 11).

Model: "model_14"

Layer (type)	Output Shape	Param #	Connected to
input_30 (InputLayer)	[(None, 100)]	0	
input_29 (InputLayer)	[(None, 1)]	0	
dense_29 (Dense)	(None, 401408)	40542208	input_30[0][0]
embedding_18 (Embedding)	(None, 1, 76)	152	input_29[0][0]
leaky_re_lu_24 (LeakyReLU)	(None, 401408)	0	dense_29[0][0]
dense_28 (Dense)	(None, 1, 3136)	241472	embedding_18[0][0]
reshape_23 (Reshape)	(None, 56, 56, 128)	0	leaky_re_lu_24[0][0]
reshape_22 (Reshape)	(None, 56, 56, 1)	0	dense_28[0][0]
concatenate_10 (Concatenate)	(None, 56, 56, 129)	0	reshape_23[0][0] reshape_22[0][0]
conv2d_transpose_8 (Conv2DTrans	(None, 113, 113, 128	148736	concatenate_10[0][0]
leaky_re_lu_25 (LeakyReLU)	(None, 113, 113, 128	0	conv2d_transpose_8[0][0]
conv2d_transpose_9 (Conv2DTrans	(None, 227, 227, 128	147584	leaky_re_lu_25[0][0]
leaky_re_lu_26 (LeakyReLU)	(None, 227, 227, 128	0	conv2d_transpose_9[0][0]
conv2d_16 (Conv2D)	(None, 224, 224, 1)	2049	leaky_re_lu_26[0][0]

Total params: 41,082,201
Trainable params: 41,082,201
Non-trainable params: 0

Figure 11. DCGAN Layer Architecture

Utilizing a GPU on Google Colab, I trained the network on the 'a' characters for 100 epochs, and the 'j' characters for 300 epochs. At first glance, the 300 epoch fit cycle produced visually better results with no signs of mode collapse (Figure 12).

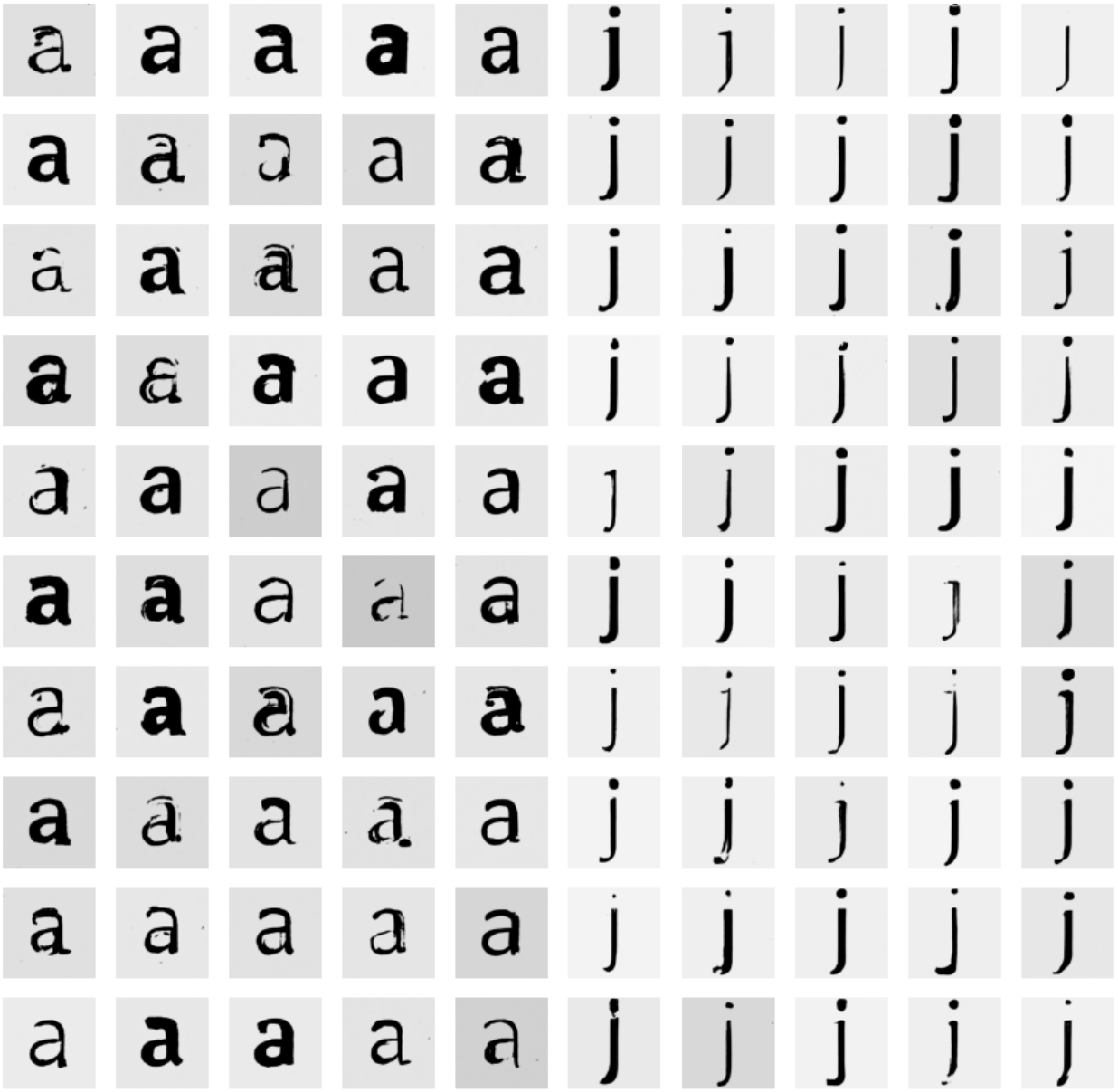

Figure 12. Representative ‘a’s after 100 epochs of training and ‘j’s after 300 epochs

The next step was to condition a GAN architecture on each class and use one model to generate for both the ‘a’ and the ‘j’ classes. This architecture was largely

similar, but included an embedding layer to amount for the classes which was appended to the output layer (Figure 12).

```
Model: "model_82"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_112 (InputLayer)          [(None, 100)]        0
__________________________________________________________________________________________________
input_111 (InputLayer)          [(None, 1)]          0
__________________________________________________________________________________________________
dense_111 (Dense)               (None, 401408)       40542208    input_112[0][0]
__________________________________________________________________________________________________
embedding_55 (Embedding)        (None, 1, 76)        152         input_111[0][0]
__________________________________________________________________________________________________
leaky_re_lu_137 (LeakyReLU)     (None, 401408)       0           dense_111[0][0]
__________________________________________________________________________________________________
dense_110 (Dense)               (None, 1, 3136)      241472      embedding_55[0][0]
__________________________________________________________________________________________________
reshape_83 (Reshape)            (None, 56, 56, 128)  0           leaky_re_lu_137[0][0]
__________________________________________________________________________________________________
reshape_82 (Reshape)            (None, 56, 56, 1)    0           dense_110[0][0]
__________________________________________________________________________________________________
concatenate_55 (Concatenate)    (None, 56, 56, 129)  0           reshape_83[0][0]
                                                                 reshape_82[0][0]
__________________________________________________________________________________________________
conv2d_transpose_54 (Conv2DTran (None, 113, 113, 128 148736      concatenate_55[0][0]
__________________________________________________________________________________________________
leaky_re_lu_138 (LeakyReLU)     (None, 113, 113, 128 0           conv2d_transpose_54[0][0]
__________________________________________________________________________________________________
conv2d_transpose_55 (Conv2DTran (None, 227, 227, 128 147584      leaky_re_lu_138[0][0]
__________________________________________________________________________________________________
leaky_re_lu_139 (LeakyReLU)     (None, 227, 227, 128 0           conv2d_transpose_55[0][0]
__________________________________________________________________________________________________
conv2d_83 (Conv2D)              (None, 224, 224, 1)  2049        leaky_re_lu_139[0][0]
__________________________________________________________________________________________________
model_80 (Functional)           (None, 1)            4400921     conv2d_83[0][0]
                                                                 input_111[0][0]
==================================================================================================
Total params: 45,483,122
Trainable params: 41,082,201
Non-trainable params: 4,400,921
```

Figure 12. Conditional DCGAN Layer Architecture

After 50 epochs of training, the paired character output appeared realistic and convincing to the human eye. Though the model could benefit from more training, the fidelity was was acceptable and could easily be mistaken for printed word—with a bit printer smear (Figure 13).

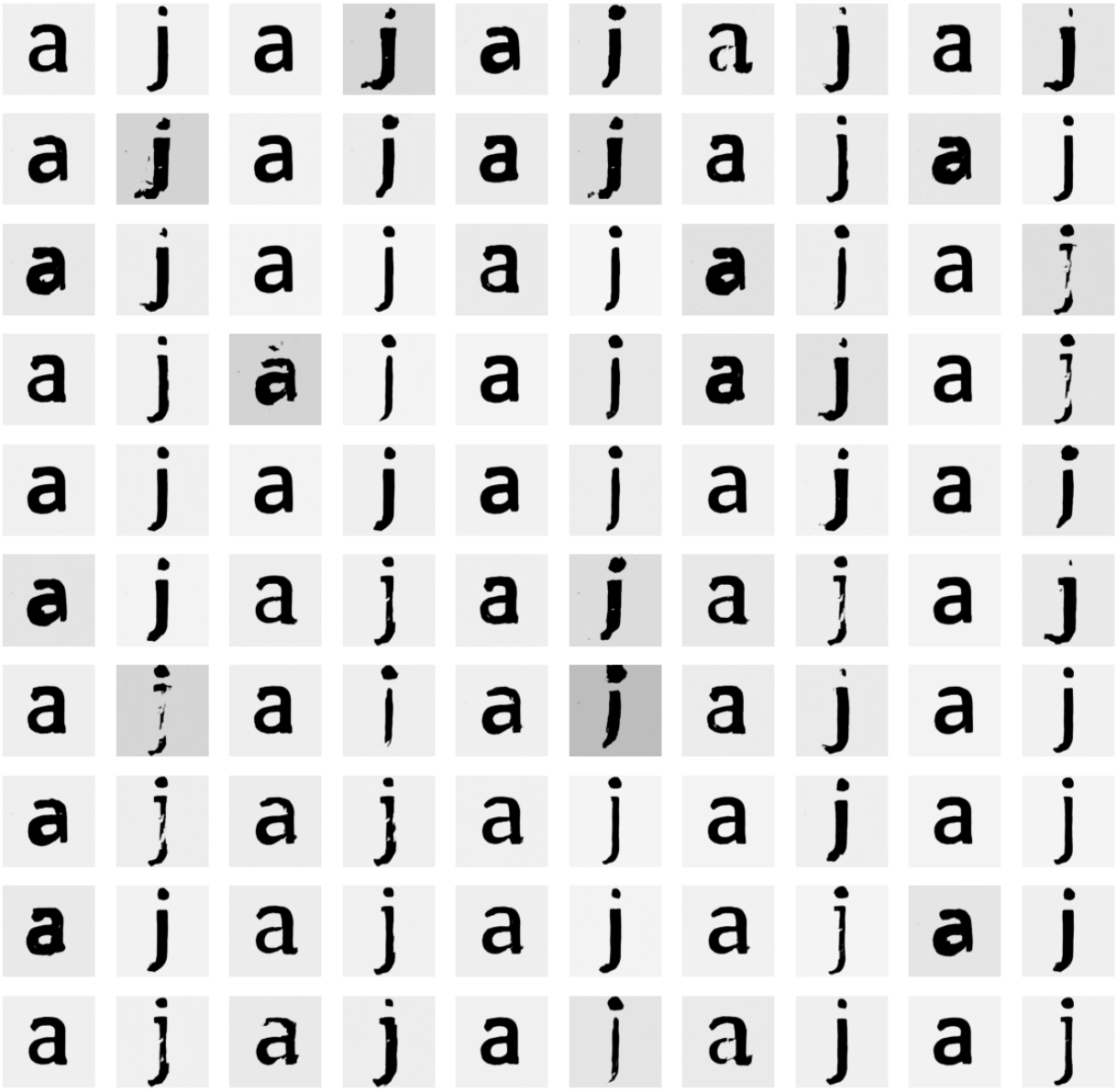

Figure 13. Conditional DCGAN Representative output after 50 epochs

Performance and Shortcomings

After further inspection, neither the ‘a’s nor the ‘j’s from either DCGAN represent the full diversity of the class (as observed in the EDA phase). The output shows a lack of representation for the class features including italic characters and alternative letter forms.

Troubleshooting

Though the results after 50 epochs are likely good enough to accomplish the goal of generating a cohesive typeface, there appears to be a lack of variety when compared to latent spaces of these characters seen during EDA. In order to rectify this discrepancy, I first inspected and verified that the data feeding into the model were diverse and representative of each image class. Next, I inspected the performance of the discriminator as a solo model outside of the DCGAN, and it performed as expected.

My final effort was to optimize the DCGAN architecture for additional variance in the generated images, I adjusted the constraints on the generated latent points that fed the generator. It was observed that the original DCGAN was generating points in a very narrow band of each character's latent space. By adding a random multiple (between -25 & 25) to the generated points, the optimized model was able to generate 'a' and 'j' characters that display much more diversity in shape, thickness, slant and overall style (Figure 14). Though this produced a noticeable improvement, after discussion with my mentor, this is likely not a sustainable solution as I'm ultimately just scaling randomness with more randomness. My only remaining hypothesis is that a scaling issue exists somewhere else in the model—potentially at the pixel level of the images.

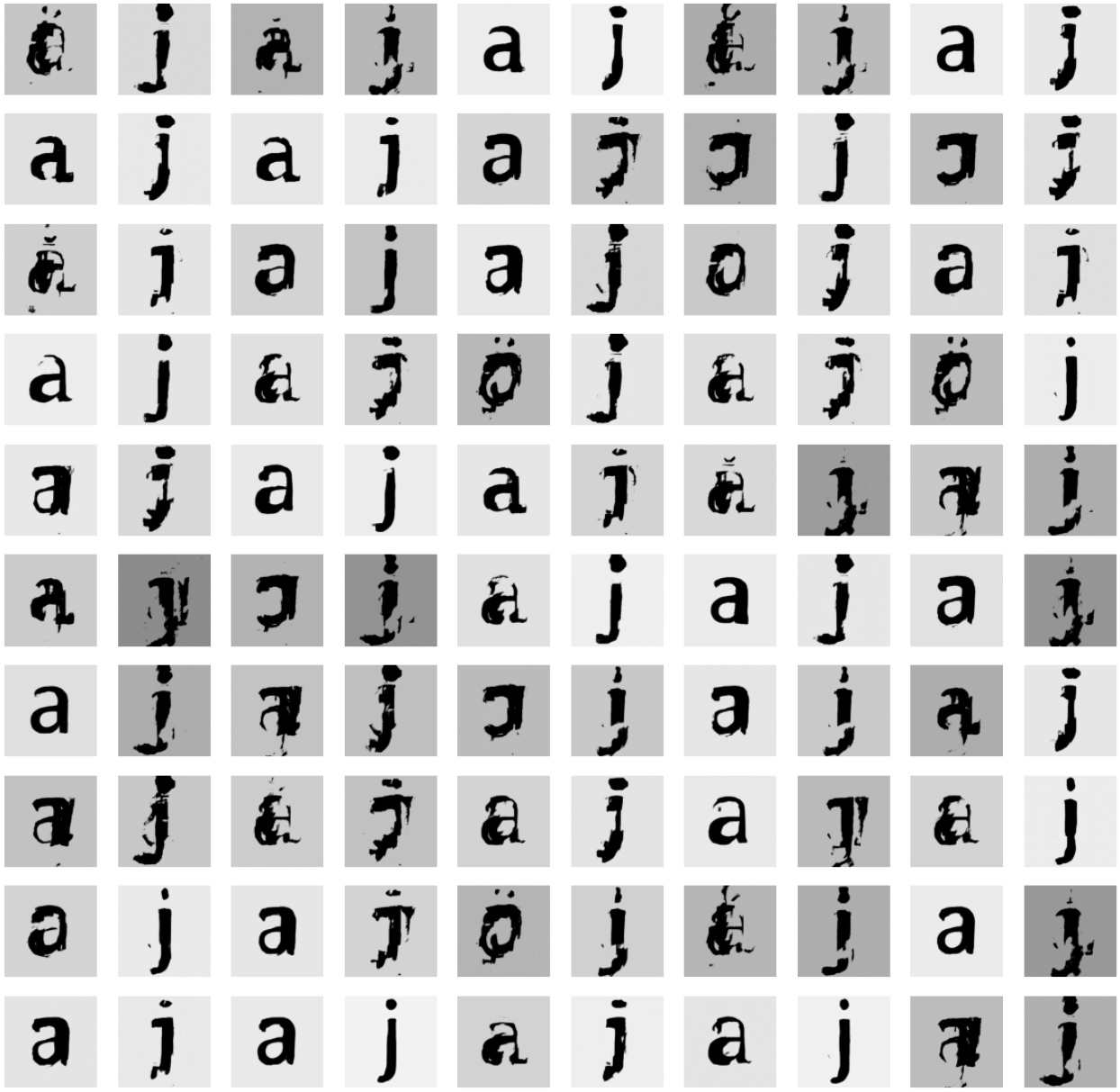

Figure 14. Representative output of optimized Conditional DCGAN after 50 epochs

Conclusion & Future Research

The goal of this work was to gain a better understanding of type and the latent representations of characters within known fonts. Ideally building enough understanding to generate a new cohesive typeface not known by any name. While the results are producing convincing characters, more research needs to be conducted to reproduce more of the known variance in each character and ultimately to generate a wider variety of typefaces.

The existing implementation is transferable to all 62 character classes. Though the size of the dataset holds some computing constraints (2 epochs in 24 hours on an Google Colab GPU), the initial 62 class output suggests it would improve at the same rate as the 2-class ‘a’ and ‘j’ output (Figure 15).

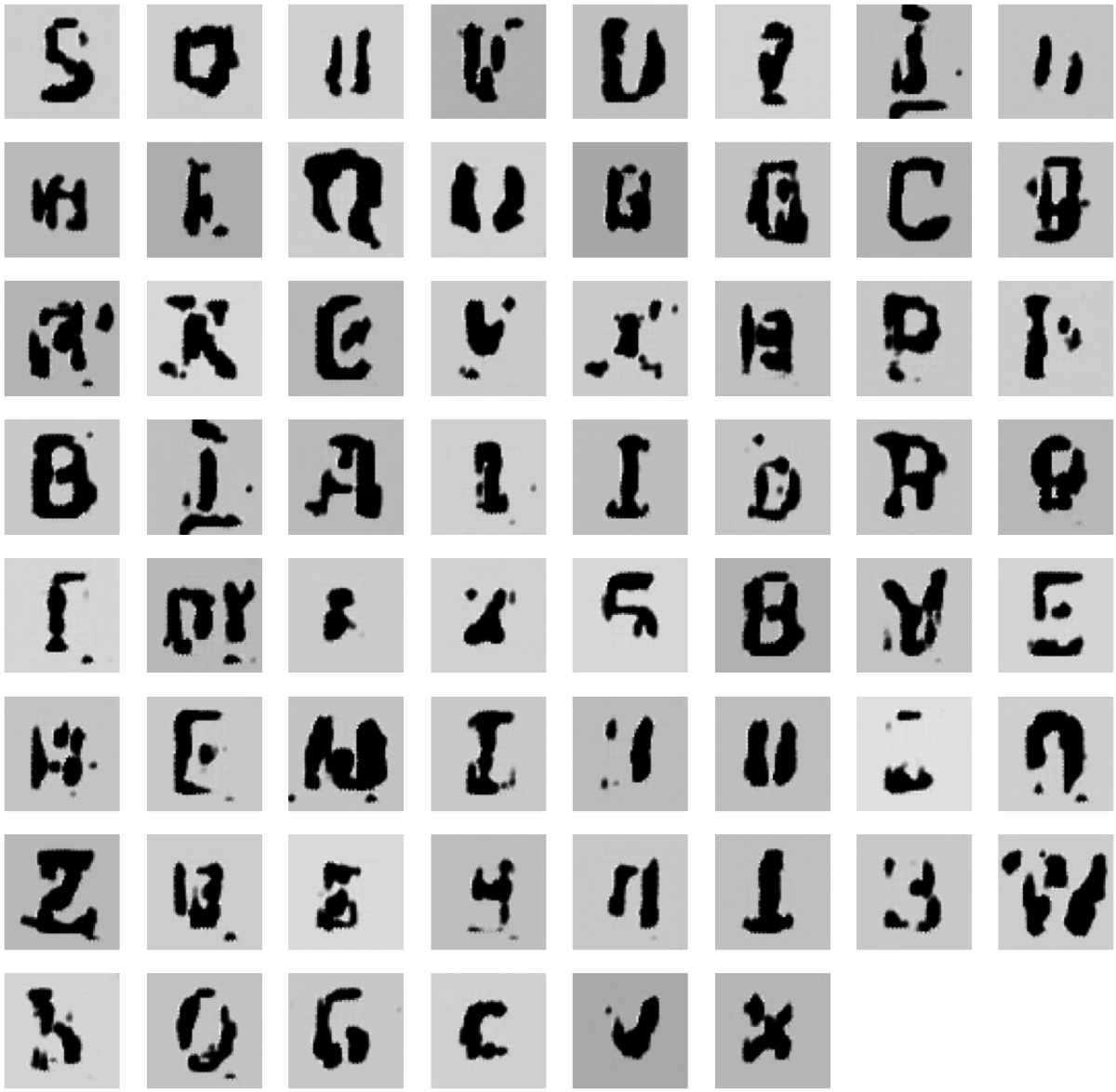

Figure 15. Representative output of 62 class Conditional DCGAN after 2 epochs

With the variety issue solved and with additional time, I'd like to take the image processing a bit further to smooth out the image of each character, then transfer the generated images into outside software to create a usable, new-to-the-world typeface usable in any common word processing tool.

In addition, I'd like to extend the latent space understanding to help designers control the output of their work. Instead of selecting fonts by name, a tool that allows them to generate the font or typeface they're looking for: something between two named fonts for example, or between a bold and an italic, would save designers a large amount of time during projects.