Typeface Space: Using Neural Networks for Font Selection

by
Samuel Magid

Professor Jeannie Albrecht, Advisor
Professor Mark Hopkins, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

April 25, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

According to Karen Cheng's 2006 book *Designing Type,* there were—in 2006—likely over 300,000 individual typefaces available to users. Today, almost twenty years later, that number is no doubt even higher; however, the most common font selection tool across all of the major word processors (Microsoft Word, Apple Pages, Google Docs) and many more is the same interface which existed on the earliest computers to support multi-font word processing: a scrollable, linear list of fonts to choose between. A few things have been improved on this basic model (displaying font names in their own typeface and list alphabetization), but the fundamental aspects of this interface have remained unchanged. If users have access to a number of typefaces which is many orders of magnitude greater than the days of early word processing, this is an issue. Users cannot be expected to navigate through such a large number of fonts; scrolling through 300,000 items is not a reasonable ask. Moreover, this list based interface does not align with the typical needs of a user: when searching for fonts, users often have a particular style in mind (professional, casual, festive), and this basic list-based interface does not incorporate any notion of style as part of the search. After over forty years of little development in the area of font selection interface—meanwhile the numbers of available fonts has increased exponentially—there is a need for better font selection tools, particularly interfaces which take into account meaningful aspects of typeface style.

1.2 Goals and Contributions

This thesis attempts to address this problem. We hypothesize that training Autoencoder-like neural networks on font image data will yield meaningful style encoding vectors which quantify different aspects of typeface style, upon which useful style-based font selection tools could be built. We implement three neural network models, all based on the original Autoencoder model, which attempt to encode typeface style in the intermediate vector representation between the encoding and decoding steps of the Autoencoder. Using the style encodings from the most effective model, we build a novel font selection tool webapp on an end-to-end system which provides the selector tool with spatial

data from the model. Evaluating both our model space against attribute categories provided by Google Fonts, we find that our models effectively encode certain aspects of style, with similarity scores within style categories increasing as model complexity increases.

Talk about the user study

Ultimately, we find that the models we have implemented, especially our final model adapted from Srivatsan et al. and trained on our full dataset, effectively encode many aspects of typeface style. We additionally see that—as shown by our novel font selection interface—these numeric style encodings can be used to create useful style-based font selection interfaces. This research contributes to the field of typeface style inference and suggests that further work can be done towards the issue of creating effective and simple style-based font selection tools.

1.3 Roadmap

This thesis document is organized as follows. Chapter 2 explores some of the background and history of font selection tools—looking at early font selection interfaces as well as a couple of recently developed language-based tools—necessary explanation of Autoencoders and Autoencoder-like models, and finally some previous work on font selection and font inference. Chapter 3 details our three models (Basic Autoencoder, Style Transfer, and our model adapted from Srivatsan et al.), explains the mechanism for obtaining the model style encodings for use in our user tool, discusses the design choices for our novel font selector interface, and details the end-to-end system upon which the tool is built. Chapter 4 includes a quantitative evaluation of font distance similarity based on attribute categories from the Google Fonts library, as well as a user evaluation of our font selection tool. Chapter 5 concludes this document by summarizing our findings, enumerating a few areas for potential future work, and explains some of the lessons I have learned in the process of this extended research project.

Chapter 2

Background

2.1 Font Selection

This section explores some important background information about the history and current state of font selection tools. Comparing one of the earliest font selection interfaces, on the 1981 Xerox Star, to the typical font selection interfaces on Google Docs, Microsoft Word, and Apple Pages, it is evident that little progress has been made in the way we envision the selection of fonts. Although these tools have remained mostly unchanged for the past several decades, there has been some recent innovation in typeface selection interfaces, and we explore these new implementations—specifically the new language-based font selection models of the Google Fonts website and Canva, a popular online graphic design tool; we additionally discuss the difficulty of creating language-based font selection tools. Lastly, we provide an analogue between font selection interfaces and the much more diverse field of color selection tools, which represent another common selection in graphical design.

2.1.1 Lack of Development in Font Selection Interfaces

Surprisingly little progress has been made in the past several decades on the issue of user font selection. Figure 2.1 shows the font selection interface on the 1981 Xerox Star, one of the earliest personal computers to include a multi-font word processor. This interface will probably look familiar to a modern user: it provides control over font size; bold, italic, underline, and strikethrough toggles; superscript and subscript options; and, importantly, a scrollable list of fonts to choose between. Compared to the modern font selection interfaces shown in Figure 2.2, the only significant difference over this 44-year gap is the alphabetization of the font selector list; otherwise, these interfaces are practically identical. While there are many beneficial aspects of this interface—control over size, boldness, and other useful dimensions of typeface—the list-based typeface selector has become increasingly unfit for the task of font selection. Whereas the Xerox Star provided its users with a small number of typefaces to choose between, today the number of individual typefaces available to users (although impossible to enumerate exactly) almost certainly exceeds 300,000 [4]. It is no longer feasible for a user to scroll through this quantity of typefaces when making a font

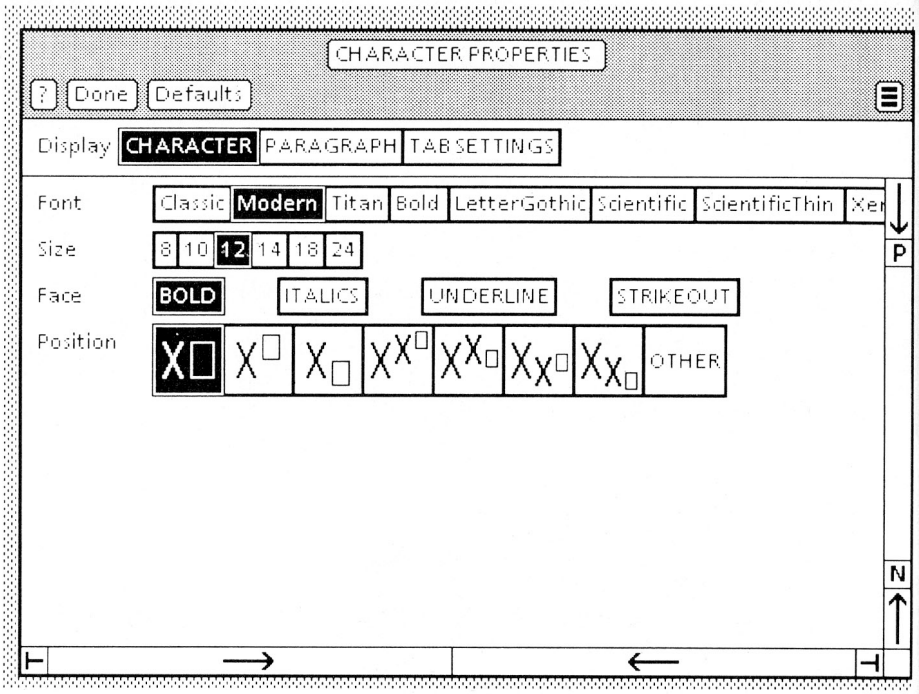

Figure 2.1: Font selection interface in Xerox Star (1981)

selection. Additionally, this inteface provides no control over style: even in an alphabetized list, there is no particular relationship between nearby fonts or any capability for style-based search. This increasingly limited user tool leads users to find a few fonts they like and stick with them, rather than explore the wide variety available to them. Without a better tool for navigating the many dimensions of typeface style, this will remain the case.

O'Donovan et al. [8] lists several reasons for the difficulty of developing font selection tools. The first issue, as discussed above, is the sheer number of available fonts. "Most computers are now equipped with hundreds of fonts," they note, while online resources provide access to hundreds of thousands. Another issue is a lack of obvious ways to categorize fonts in a manner which corresponds to user goals. While there exist broad categories like Serif, Sans Serif, and Handwritten, these must be manually designated on a per-font basis, and they are not necessarily helpful to every user. A college student, for example, might know to choose a Serif font for their paper to convey an academic mood—or, more practically, to fulfill certain departmental design expectations—however another user, looking to design a new logo for their coffee shop, might not find the distinction between Serif, Sans Serif, and Handwritten typeface particularly useful or informative. A graphic designer, who has studied typeface design for years and has relevant experience, might feel confident in handling these categorizations, but most users do not have a more amateur background. Typical users lack the necessary tools, given the current state of font selector interfaces, to properly consider the wide range of diverse typefaces and confidently choose the right font—one of the most fundamental decisions in effective text-based graphical design. Finally, different users vary in their font selection goals.

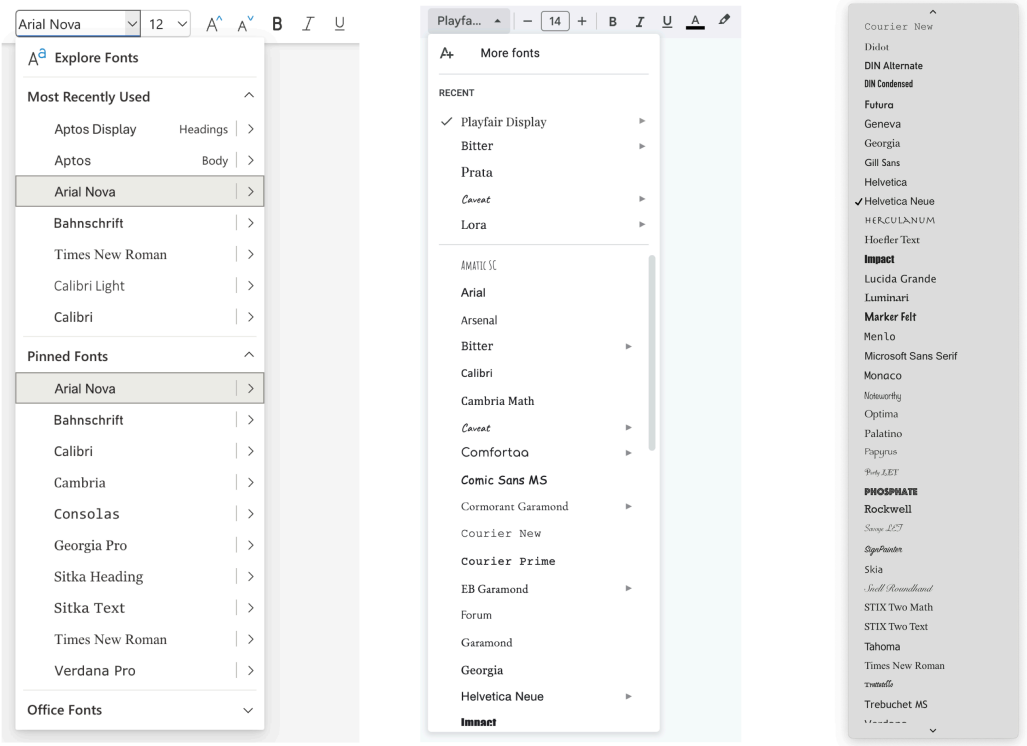

Figure 2.2: Current font selection interfaces in Microsoft Word, Google Docs, and Apple Pages

One user may be looking to identify the font they saw on a store sign or a brochure—or to find a free-to-use font which is similar to a commercial one they have identified. Another may be looking to match a particular mood, or to choose a font that fits well with the rest of their document. A third may simply be exploring a large set of fonts like Adobe TypeKit or Google Fonts, curious to find new, exciting typefaces. O'Donovan et al. argue—and we agree—that current methods of font selection fall short on these issues and more. Given the growing number of fonts available to the modern user, a better, more useful system for typeface selection is long overdue.

2.1.2 Font Selection Innovation

There has been some limited progress, in recent years, in the field of font selection interfaces. Specifically, both Google Fonts and Canva, a popular online graphic design tool, have experimented with language-based font selection tools. Both of these interfaces break from the common list-based font selection tool, but both also have some significant drawbacks and flaws. This section will explore these two novel font selection interfaces and their respective contributions.

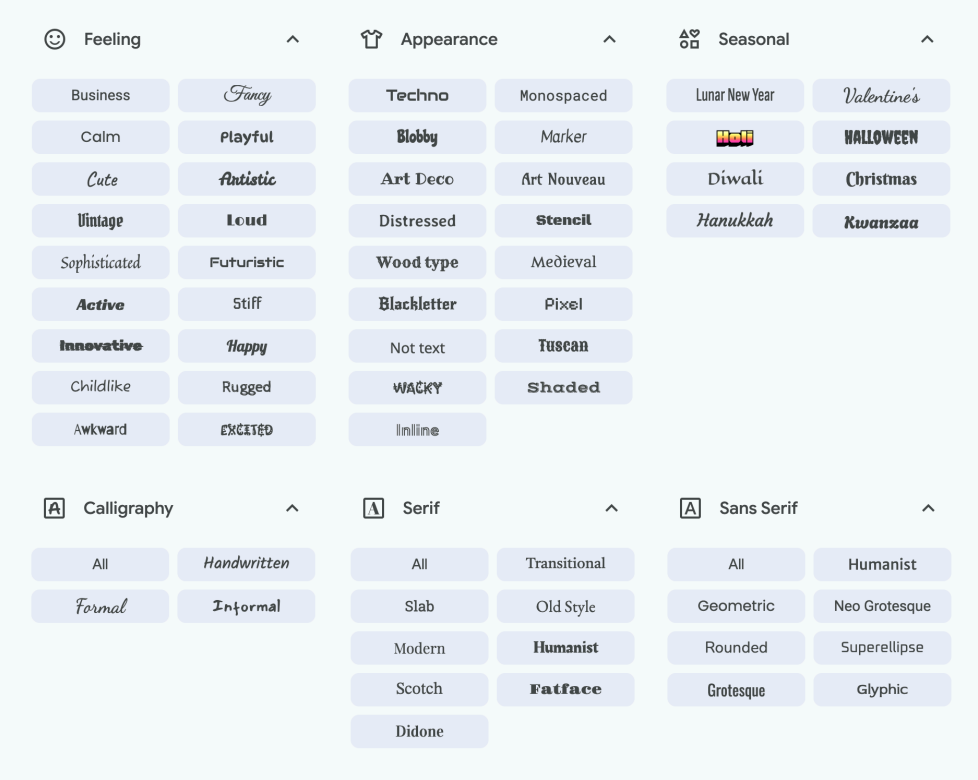

Figure 2.3: Some typeface categorizations in the new Google Fonts interface

The new font selection interface on the Google Fonts website[1] was released in early 2025. Whereas Google previously sectioned their fonts into 5 broad categories (Display, Handwriting, Monospace, Serif, and Sans Serif), their new interface introduces a much larger set of typeface categories, broken into broader parent categories. In the "Feeling" category, for example, users can filter "Happy" fonts, "Calm" or "Playful" fonts, and "Childlike" or "Awkward" fonts. The new interface includes appearance categories like "Techno" or "Art Deco," and also holiday categories like Halloween, Hanukkah, Christmas, and Holi.

Google Font's categories are impressive and certainly break with the basic list-based font-selection interface; however, the tool has several drawbacks. For one, the user is forced to choose between discrete categories, rather than an open-ended text input, limiting them in their selection process. Additionally, while many of the categorizations seem fairly accurate, some are questionable (such as the "Christmas" fonts in Figure 2.4). Most importantly, however, Google has not incorporated this new font selection tool into their main Google Suite applications (Google Docs, Google Slides, and Google Sheets) where the majority of their users are choosing fonts. This new font selector tool, available only from a separate website, is an interesting experiment in language-based font selection, but not much more than that.

Canva, a popular online graphic design tool, has also experimented with a language-based font selection tool: their main design interface allows users access to select fonts via a text input field. A

[1] https://fonts.google.com

Figure 2.4: Some of the "Christmas" fonts in Google Fonts do not seem very Christmassy

user could type "Modern" and they would be presented with a wide-variety of modern-style fonts. However, the tool is somewhat deceiving: while the input prompt appears open-ended, the tool actually only works for a small set of keywords; for most text input, it will either yield no results or simply return fonts whose name contains that keyword. (See Figure 2.5 for some usage examples.) For unseen inputs like "Professional," "Cheerful," "Ancient," and "Lightweight" the tool returns no results at all. As with the new Google Fonts category-based selector, this font selection interface seems appears more of an interesting experiment than a user-ready tool.

2.1.3 Issues with Language-Based Models

Both of the above tools introduce language-based models for font selection, one with clearly delineated style categories (Google Fonts) and the other a seemingly open-ended tool which, in reality, allows only a limited set of keywords as input (Canva). The direction of this approach, however, is not a bad one. Language is one of the ways in which humans fundamentally conceive of the world, including with respect to visual style. Especially given the popularization of Large Language Models (LLMs) and chatbots, there is certainly an open space for innovation with language-based font selection. However, there are a couple key problems with creating these language-based font selection models. First of all, visual style is quite subjective: one user might find a certain font "wacky" while another might find that font "sad" or "disgusting." A font which one user finds "professional" another might find "playful." Building a language-based model for font selection would therefore have to account for some amount of user subjectivity in its recommendations. Secondly, there is a relative lack of datasets which connect typefaces to language-based style characteristics. Shaikh et al. [10] perform an online study with hundreds of participants to generate a dataset of only 20 fonts and 15 style adjectives. O'Donovan et al. [8] generate a larger dataset of 200 fonts and

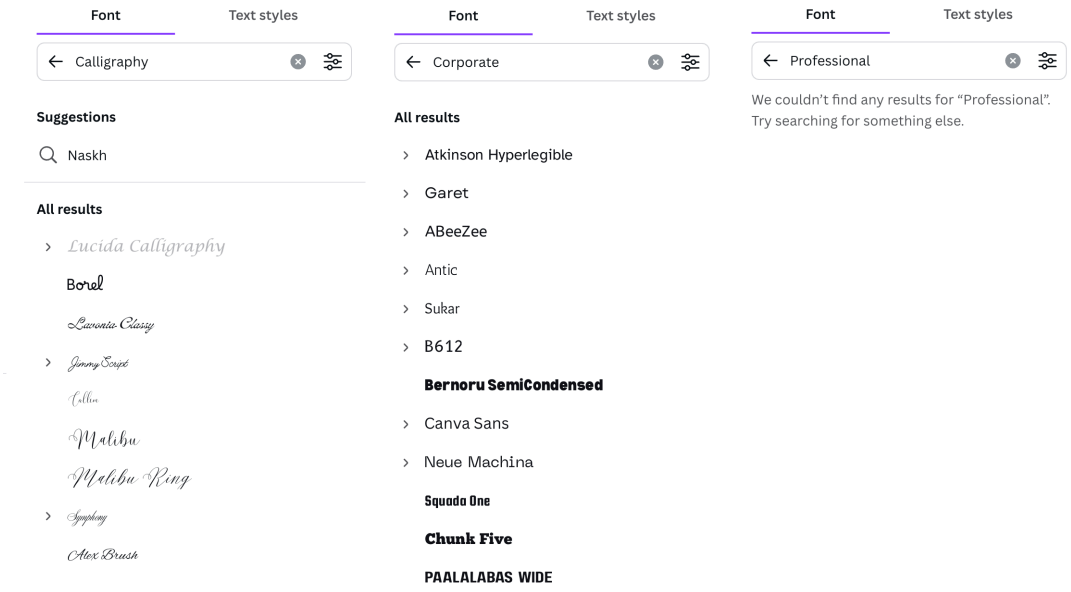

Figure 2.5: Example usages of the Canva text-based font selector tool

31 style adjectives, but this is still relatively small when compared with the hundreds of thousands of available fonts and the many possible dimensions of style. Using the new style characteristics in Google Fonts and Canva (accepting some questionable categorizations) would provide more data to work with, but it is still unlikely that this would be sufficient to train an LLM for style-based font suggestion. Some inference could be performed to expand this data and predict attributes, but a larger human-labelled dataset would most likely be necessary to train this kind of model. For the scope of this project, we mostly avoid the issue of language-based font selection, and rather focus on the use of unsupervised neural models in building useful style-based font selection tools.

2.1.4 An Analogue: Color Selection

A useful analogue when considering the issue of typeface selection is another common issue in graphic design situations: color selection. The field of color selection has produces a much wider range of selection interfaces, which suggests a potential for a much more diverse set of font selection tools. Figure 2.6 shows a common, basic color selection tool containing several ways to interact with its many available colors: sliders, numeric input, and a 2-dimensional gradient. For historical reasons and a need for standardization, computer color is split into a 4-dimensional basis called RGBA: red-value, green-value, blue-value, and transparency-value. The first three are based on a hexadecimal scale, allowing values between 0 and 255, while the transparency value is constrained between 0 and 1. By splitting color into a multi-dimensional basis, users have an intuitive, wide-ranging control over the color selection process. This is much more useful than, say, selecting from an alphabetized list of colors (Apricot, Aquamarine, Baby Blue, Canary Yellow...), which—similar to an alphabetized

list of fonts—which would not provide users with a very meaningful way to navigate the dimensions of color. There are many other popular tools for color selection: Adobe Color, for example, utilizes a popular color-wheel tool, and additionally includes features to change color basis (CMYK and RGBA are the most common color bases, but other more obscure ones exist), pick a set of colors based on a particular harmony (Monochromatic, Triad, Complementary, e.g.), and save colors to a library for later access.

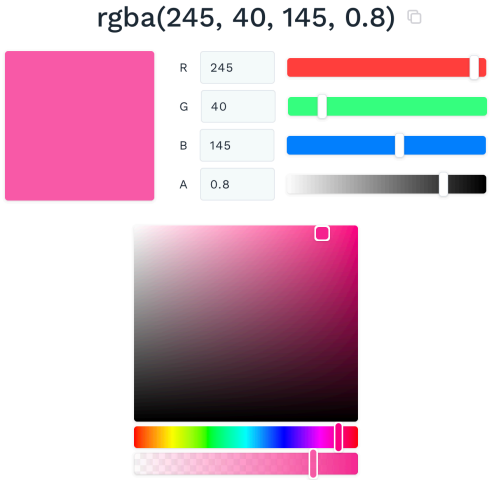

Figure 2.6: A basic color selection interface with gradient, numeric, and slider controls

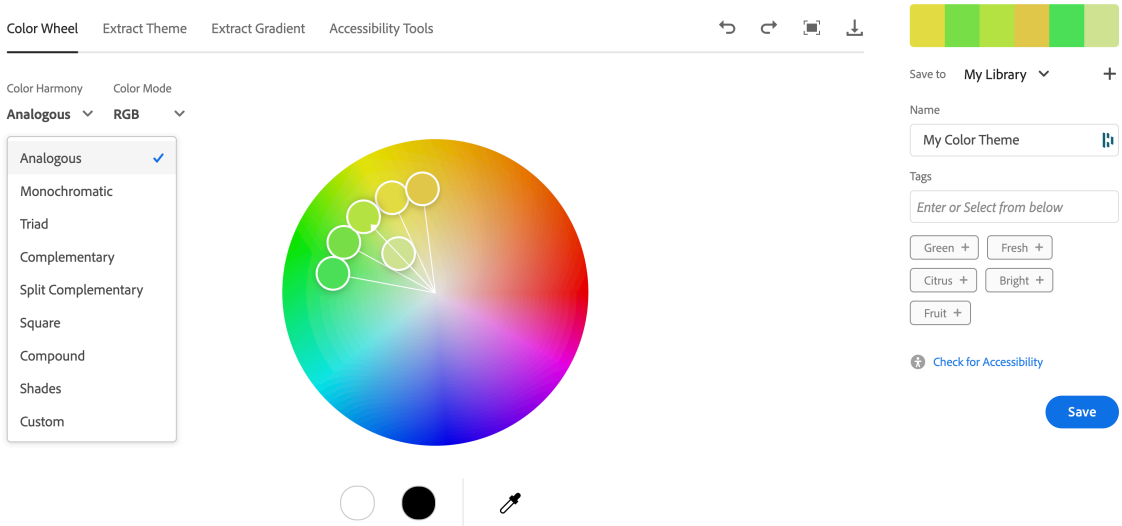

Figure 2.7: Adobe Color selection interface, with a color wheel, multiple color bases, color harmony-based selection, and the ability to save colors

The wide range of diverse color selection interfaces and color bases suggests a potential for similarly-diverse, useful selection tools for typeface which provide control over the many dimensions

of typeface style. While typefaces and their style cannot be quantified as easily (or objectively) as color, their style *can* be quantified—as we will show in subsequent chapters—and building selection tools around quantitative style dimensions would allow users much greater control in typeface selection, similar to the high level of control users already have in color selection. Imagine, perhaps, a gradient of fonts, or sliders which control different aspects of font style. These ideas are not farfetched: as our research will show, neural networks—specifically autoencoder-like neural network models—can be used to effectively condense quantitative typeface style encodings from font image data, which provides a basis upon which to build powerful style-based font selection tools.

2.2 Autoencoders

As previously stated, we hypothesize that neural networks might provide a useful foundation for font selection tools by generating typeface style encodings. More specifically, we focus on autoencoder and autoencoder-like models in our neural network design. This section, therefore, provides necessary background on the autoencoder model and why autoencoder-like neural networks might provide a useful foundation for creating style-based font selection tools.

2.2.1 Autoencoder Model

The autoencoder, originally proposed by Rumelhart et al. [9], is a specific type of neural network trained to exactly reconstruct its own input data. The model is composed of two parts: an encoder, which transforms the input data to an intermediate representation (usually smaller than the input data) through a series of linear and nonlinear layers; and the decoder, which transforms the intermediate representation back to the original input size. By minimizing the loss of this neural network during training, the encoder learns to optimally condense the input data for later reconstruction, and the decoder learns to optimally reconstruct that data based on the intermediate representation generated by the encoder.

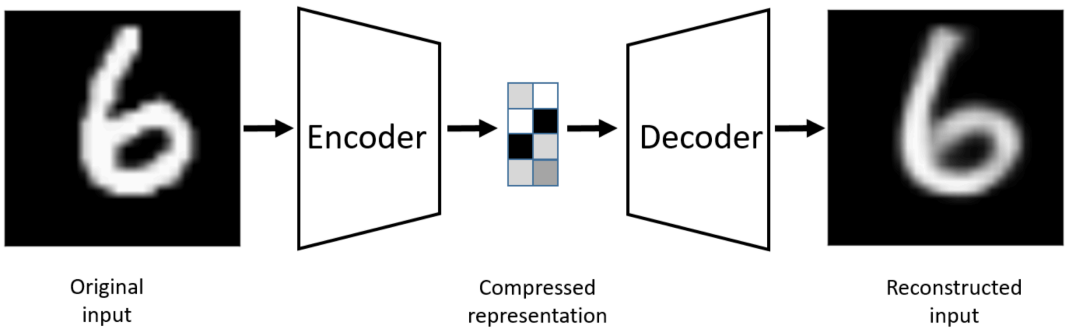

Figure 2.8: Basic autoencoder model applied on MNIST data

While the autoencoder model may seem of little use—why would I want to reconstruct data which I already have?—the autoencoder is actually a highly practical model. The main goal of an

autoencoder, rather than the final reconstructed output, is to distill a meaningful representation of the input data—the representation located between the encoder and decoder components—which might represent something intrinsic about the data. In order for the decoder to be able to accurately reconstruct input data which it has never seen, the encoder part of the model must encode a particularly useful intermediate encoding, which represents enough important aspects of the input data for the decoder to use in decoding. If the encoder does not provide a meaningful representation of the input data, the decoder cannot effectively reconstruct that data. Thus, the autoencoder's intermediate representation—between the decoder and encoder—tends to hold useful and meaningful encodings of given input data after sufficient training. In our research, we leverage these encodings as our basis for typeface style. In their well-known chapter on autoencoders, Bank et al. write:

> ...the goal of autoencoders is to get a compressed and meaningful representation. We would like to have a representation that is meaningful to us... [2]

In order to create these meaningful representations, however, some some steps must be taken to avoid simply learning the identity function. If the model were allowed, it would optimally learn to encode exactly the same data as given as an input, and decode by simply returning that data. However, this would not provide a useful or condensed representation. In order to avoid learning the identity function, the autoencoder model usually includes a bottleneck—meaning that the encoder must compress the data before giving it to the decoder (see Figure 2.8). Therefore, the model *cannot* simply learn the identity function and must learn to produce a useful, condensed version of the data. This intermediate representation is usually the desired output of an autoencoder model. In our case, the purpose of training an autoencoder model to reconstruct font data is not the actual reconstructed images, but rather the intermediate encodings which the autoencoder produces. Importantly, while a bottleneck is the most common technique to achieve this effect, other methods such as adding Gaussian noise can be used instead of or in addition to a bottleneck to achieve this regularization.

2.2.2 Variations on the Autoencoder

There have been many variations on this basic autoencoder model to prioritize different model goals. While the basic autoencoder is unsupervised, it is possible to feed additional data into the autoencoder, such as data labels, in order to coerce the model to ignore these aspects of the input data in the construction of an intermediate representation. (In our research, we employ this method to disentangle *letter* meaning from *style* meaning.) Another alteration of the original autoencoder is the variational autoencoder (VAE) model introduced by Kingma et al. [6], which uses probabilistic distributions to improve autoencoder performance, especially with respect to generative tasks. Rather than encoding an explicit intermediate encoding, VAEs encode the parameters of a multi-dimensional Gaussian distribution which represents the probabilistic space of the intermediate encoding. The decoder then samples randomly from the distribution and proceeds with the decoding task. This probabilistic model is especially useful in generative tasks, when the goal is to generate new data, but the continuous latent space encoded by VAEs can also be used, as in our case, to generate meaningful data representations such as style encodings.

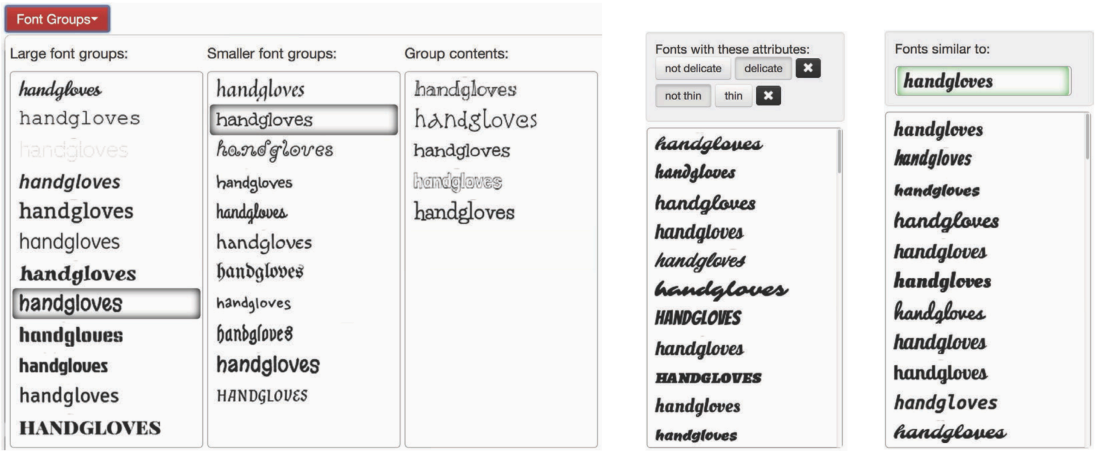

Figure 2.9: Group Interface, Attribute Interface, and Search-By-Similarity selection tools from O'Donovan et al.

2.3 Previous Work

There has been previous scholarly work to create better font selection interfaces, especially based around font inference models. This section explores a few of these models, one based on crowdsourced data, and the others on neural inference models. Both of these approaches are useful and have informed our research, but the neural inference models have particularly influenced the models we have built. Particularly, we adapt the Srivatsan et al. model later in our research and use the resulting typeface style encodings to build our typeface selector tool.

2.3.1 Crowdsourced Models

O'Donovan et al. [8] proposes three novel font-selection interfaces, built on crowdsourced data from Amazon Mechanical Turk (MTurk): one based around verbal attributes such as "formal," "friendly," or "legible" called Attribute Interface; another which clusters fonts hierarchically based on visual similarity, called Group Interface; and a third, to be paired with the other two methods, which provides users with a list of similar font to the current selection, called Search-By-Similarity (see Figure 2.9). The researchers build these models on crowdsourced data collected through MTurk: they prompted users to decide on questions such as "Which of these two fonts is more strong?" or "Which of these fonts is more silly?" to collect attribute data, and asked questions like "Which of these two fonts—Font B or Font C—is more similar to Font A?" in order to build similarity data. The Group Interface model splits fonts into large categories based on similarity, creating a tree-like font selection tool where users can progressively narrow down their font selection task. The Attribute Interface, similar to the language-based Google Fonts and Canva interfaces, attaches fonts to adjective descriptors, which is helpful as humans tend to conceptualize style based on verbal descriptors. Finally, they use the similarity data to create their Search-By-Similarity tool, to be

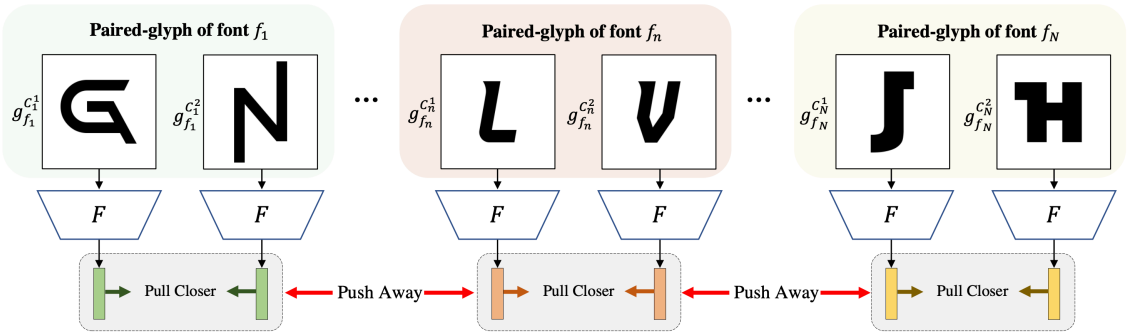

Figure 2.10: Paired-glyph matching in Cho et al.

used in conjunction with the other two methods: given a selected font, which fonts are most similar according to other users? This provides an important aspect of typeface selection: if a user is looking to select a font based on style characteristics, it is useful to see other fonts that are similar according to other users. All of these tools are potentially useful ways to navigate typeface selection by means of style, but it should be noted that basing these attributes and similarity scores on crowdsourced data means that these tools may not align with every user's subjective sense of style. O'Donovan et al. additionally conducted user studies, also on MTurk, presenting users with either font matching or design tasks to evaluate their interfaces against a baseline font selector tool. The researchers found positive results for their novel font selection tools: participants were three times more likely to succeed in a font-matching task using any of the three proposed interfaces compared with a basic list-based interface, and they also found a small statistically significant improvement in user performance on the design task between their selection interfaces and the baseline.

2.3.2 Inference Models

There has also been some effort to build inference models around font character images, often with the explicit purpose of building better user-interface for font selection. Cho et al. [5], for example, builds a model with the explicit goal of generating latent space encodings of glyphs which are easily differentiable based on their font. They describe their model design as such:

> For the discriminative representation of a font from others, we propose a paired-glyph matching-based font representation learning model that attracts the representations of glyphs in the same font to one another, but pushes away those of other fonts.

Their paired-glyph matching, shown in Figure 2.10, involves selecting random pairs of glyphs and training the model to prefer a low cosine similarity (more similar) between the representations if the glyphs are characters in the same font, and a high cosine similarity (less similar) if the glyphs come from different fonts. While Cho et al. succeeds in their goal of clustering the latent space representations of glyphs by typeface, we are skeptical of their methodology: by training the model to explicitly minimize cosine similarity between characters of the same typeface, it seems quite possible that the model will not incorporate meaningful aspects of style in the model encodings. Additionally,

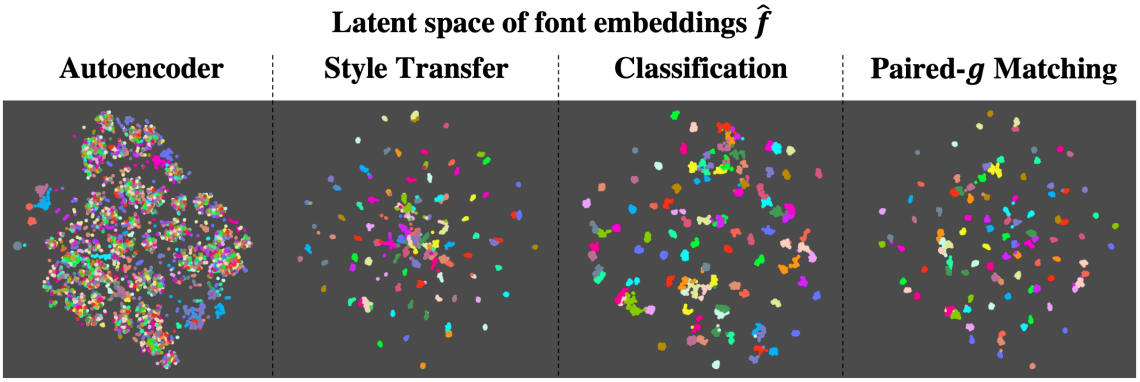

Figure 2.11: Latent space of font embeddings across model techniques in Cho et al.

to train the model and evaluate it on the same criteria (cosine similarity) seems a circular approach. Their model clearly performs well at the task on which it was trained—to discriminate between fonts—but the model space may not represent meaningful dimensions of typeface style (weight, size, serif) beyond simply separating fonts which are not the same away from each other in the style space.

One aspect of Cho et al. which is helpful for our work is their evaluation of different model technique for generating style encodings. As shown in the latent space maps in Figure 2.11, they employ three different model architectures in addition to their paired-glyph matching: the basic autoencoder model, style transfer (generating another character in the same font given an input character), and classification (predicting which font is represented in a glyph). There is an overlap in some of their model choices and ours (namely, autoencoder and style transfer), and the figure they provide is useful in visualizing the typeface-clustering effectiveness of these various models.

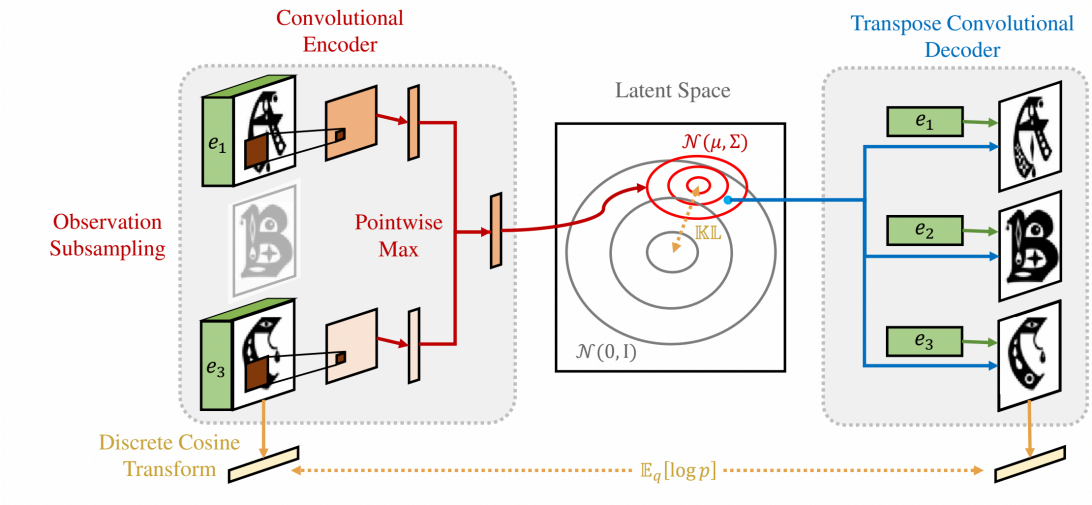

Figure 2.12: Generative process of model Srivatsan et al.

Srivatsan et al. [11] introduce an exciting novel training method based on latent probability space and a tensor factorization approach well-founded in past literature. Their model explicitly seeks to disentangle style and content—to encode the typeface style of a glyph as separate from the actual character it represents. The model architecture, shown in Figure 2.12, convolutionally encodes a probabilistic embedding of a font given its complete character glyph set (with a chosen number missing) and corresponding character embeddings, and uses that latent probability vector along with a given character embedding to reconstruct the missing glyphs. Their model is particularly effective at reconstructing glyphs, when compared to peer models, and it also succeeds against a state-of-the-art peer model when evaluated by humans on Amazon Mechanical Turk. The model additionally yields high quality style encodings, with similar fonts having more similar encodings in the model space. Figure 2.13 shows a t-SNE projection of their model latent space with "A" glyphs displayed at each centroid given k-means clustering ($k = 10$). These centroids are representative of the typeface styles existing at the given area of the t-SNE plot. Lastly, Srivatsan et al. find that, qualitatively, their model seems to effectively recreate many important aspects of character style.

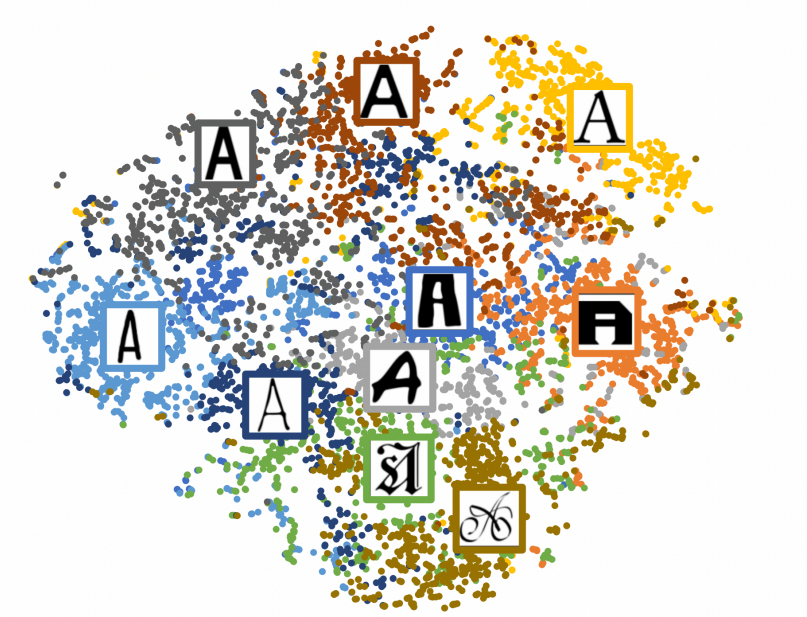

Figure 2.13: t-SNE projection of latent font variables in Srivatsan et al. with centroids

As previously stated, we chose to adapt the model from Srivatsan et al. in our research, porting it to a more recent version of Python (3.13) and PyTorch (2.5.1) and training the model on a larger dataset, comprised of the model's original training data, preinstalled Apple fonts, and the set of fonts available from the Google Fonts repository. We implement this model, along with two of our own models, and compare their respective style encodings in the following sections.

Chapter 3

Model and System Design

3.1 Data Collection

In order to train models to effectively encode the style of typefaces, it was necessary to create a dataset of characters from a variety fonts. Given the great diversity in typeface style between fonts, it was important that the dataset be large and representative enough to encompass the variety of existing typeface styles, so that the model would work effectively not just for fonts in its training set, but for unseen fonts as well.

The first important consideration was choosing the source of our data. The model proposed by Srivatsan et al. [11] was trained on the large Capitals64 dataset constructed by [1]. We also chose to use this dataset, but opted to add some additional sources of fonts. Specifically, we included the entire library of fonts from the Google Fonts repository,[1] which contains a wide range of free-to-use fonts for use in Google applications (Google Docs, Google Slides, etc.), personal websites, and for download, and we also included the 132 fonts which come preinstalled with macOS. In the latter two cases, we only considered fonts which supported English-language text, choosing to ignore typefaces in other scripts (Chinese, Arabic, Bengali, e.g.) for the scope of this research. Ultimately, we included in our dataset 10,682 typefaces from the Capitals64 dataset, 3,577 from Google Fonts, and 132 fonts from the installation of macOS (which contains many of the well-known proprietary fonts not represented in Google Fonts or Capitals64). We felt that this combined dataset ($n = 14,391$) would be both sufficiently large and representative of a wide variety of typeface styles, while still containing well-known proprietary fonts such as Times New Roman and Helvetica.

The fonts we sourced came as TrueType (.ttf) and OpenType (.otf) binary font files, both of which contain the raw data for individual typefaces or multiple typefaces in one family (Helvetica and Helvetica Light, e.g.). We used the open-source FontForge scripting package[2] to create an SVG vector image file for each character represented in each typeface, then rasterized the character vector files to 64×64 pixel images in .png format. We additionally created 1664×64 images representing the 26 capital letters (A-Z) in each typeface to fit the Capital64 data format expected by the Srivatsan

[1] https://github.com/google/fonts/
[2] https://fontforge.org/en-US/

et al. model implementation. For font files which contained multiple styles in a given font family, we treated each style as a separate typeface, since their respective style representations should be meaningfully different.

3.2 Models

Over the course of our research, we employed several model training methods. This section details the various approaches and their respective model architectures. All models were trained in PyTorch 2.5.1 and Python 3.13 running on a Bizon G7000 G2 GPU server with 2x 32-Core 2.00 GHz Intel Xeon Gold 6338 CPUs and 4x NVIDIA RTX A6000 48 GB GPUs. Generally, we trained our models until the model loss plateaued and the font reconstruction task stopped improving.

3.2.1 Basic Autoencoder

For our first approach, we adapted the original autoencoder model proposed in [9]. As previously explained, an autoencoder is a neural network trained to compress and reconstruct input data. By doing so, it can learn to generate meaningful encodings of input data; in the case of our research, we hoped to represent typeface style in these model encodings. We implemented an autoencoder model trained on our large dataset of font character images, converted into pixel-intensity matrices. After flattening the data, our model uses a series of alternating linear layers and ReLU activation functions to compress the the input $64 \times 64 = 4,096$ vectors to 6-length vectors, approximately halving the size of the vector with each linear step. The decoder, conversely, expands the data back to its original size with a series of linear and non-linear layers, ultimately converting the vectors back into 64×64 pixel intensity matrices. To compute the loss of our model, we used an MSE function computing the differences between respective pixel values between the input and output matrices. After computing this loss, we translated the pixel intensity matrices back into their original image form, in order to visually evaluate the success of our reconstructions.

Some example input and output data from our initial autoencoder model can be found in Figure 3.1. Even early in our training, the model was able to accurately reconstruct most of the input characters, although struggling more with the more complicated typeface styles. However, we quickly realized an issue with this model: given only pixel values as data, the autoencoder has no obvious ability to designate form (the style of a character, defined by its font) from content (the actual letter which is represented). For example, an A in Helvetica looks substantively different than an *A* in Comic Sans, but an A in Helvetica *also* looks substantively different from a B in Helvetica, simply because they are different letters. Our basic autoencoder does not have the necessary data to tell that Helvetica's A and B should have the same style even though their bitmap representations are very different, and that the Comic Sans *A*—whose input data actually looks more similar to the A in Helvetica—should have a different style encoding. Under this model, the six dimensions of the resulting style encoding would not only represent the variance in style between characters; it would represent the actual variance in the characters themselves. In other words, the model would learn how to represent the difference between an A and B character, but it would not be able to separate

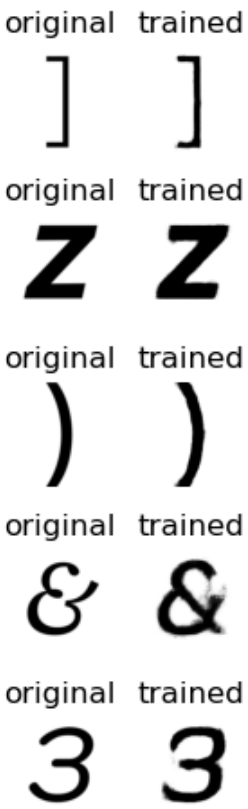

Figure 3.1: Our basic autoencoder model inputs/outputs mid-way through training

this difference from the differences in style between characters of different typefaces.

In order to create style encodings for style-based font selection, an effective model must be able to understand *character* as separate from *style.* Characters in the same font should all be understood to have one style, determined by the font itself, while two characters in different fonts should have different style encodings (which might be more or less similar depending on the similarity of their respective styles).

3.2.2 Style Transfer Autoencoder

To disentangle this character-style duality, we trained a new modified autoencoder on a different task: to recreate *other* characters in a given font. For example, given an input image of a C character, we trained our model to output a Q character in that same font. In order to provide the necessary information for the model to succeed in this task, we passed two additional vectors into the model as parameters: one representing the character of the input image, and the other representing the target character. The model would take as input a characer image as input, along with the vector embedding parameter representing the character information, and would additionally take the target character embedding as a mid-way input to provide the decoding step with the requisite

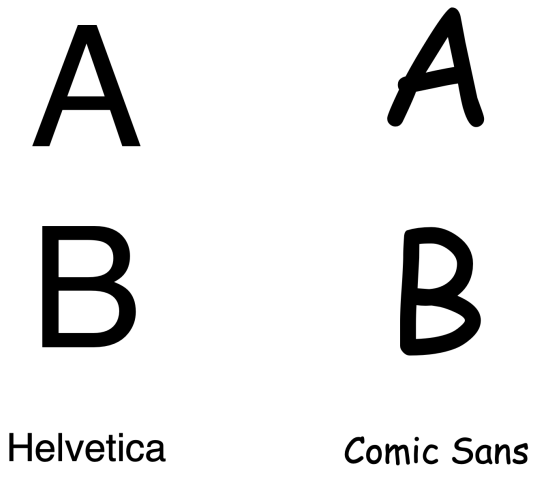

Figure 3.2: An A in Helvetica is more similar to an *A* in Comic Sans than to a B in Helvetica

information to construct the target image. By minimizing the MSE difference between the generated character (say, a Q in Arial) and the ground truth image in our dataset, we hypothesized that our model would better isolate style in its encodings. There would be no need to represent character in the intermediate model encoding, since both the encoder and decoder receive auxiliary character information.

Since this model relies on character translation, it was necessary to ensure a common character set across our typefaces. It would not make sense to train a model to translate between all the characters of two typefaces if one typeface contained a much more limited set of characters, for example missing certain symbols or only representing lowercase letters. Additionally, the auxiliary character embeddings paramaterized in the model required that there be a fixed-size character set. We originally experimented with limiting our sample size to only include fonts which contained all uppercase (A–Z) and lowercase (a-z) English characters, however in order to incorporate the Capital64 dataset—which only includes capital letters—we decided to limit our sample size only to fonts including the English uppercase letters, and trained our style transfer model only on these characters. Fonts which did not contain the 26 uppercase letters were excluded from training, but given the size and diversity of our dataset this likely did not meaningfully impact the resulting style encodings of our model.

Besides the different training tasks, there was a marked difference in our data handling techniques between our style transfer model and our basic autoencoder model described in 3.2.1. First, we had to significantly modify the data loading process: although we were working with a smaller subset of typefaces and character options (ignoring numbers, punctuation, and special characters), our dataset was in fact larger than in our original autoencoder model. Rather than dealing with every individual character in a font, we needed to handle every *pair* of characters in a font, effectively enlarging our dataset by several folds. Since we were considering 26 characters per font, our model had to train on ${}_{26}C_2 = 325$ character pairs per typeface. It was necessary to modify our dataloader to properly

handle every pair of characters (we achieved this using modular arithmetic) and to provide, for each entry in the dataset, both the input truth and output truth images, as well as the input/output characters. Since we were considering the input/output characters as data inside our model, unlike the basic autoencoder approach, we also created embedding parameters for each of the characters within our model architecture. We created a parameterized vector embedding for each character; the input embedding was concatenated with the flattened input image data before it was fed through the encoder, and the output embedding was concatenated with the intermediate vector representation between the encoder and decoder steps. Finally, we computed the MSE loss not against the input image but against the ground truth goal image representing the target character in the selected font.

Figure 3.3 shows our style transfer model part-way through training. The model receives the input **B** glyph and the input/output character embeddings (not shown), and it attempts to reconstruct the **h** glyph in the same typeface. By giving the model explicit vector representation of the input/output characters, we hypothesized that the model would more effectively isolate the style of the glyphs as separate from their content, giving us better internal model embeddings to leverage for style-based typeface selection.

Figure 3.3: Example instance of our style transfer model part-way through training

3.2.3 Srivatsan Model

Our final model, which we adapted from Srivatsan et al. [11], mirrors our previous approaches in many ways, but it additionally introduces several distinct techniques which may improve the stylistic encoding ability of the model on our dataset. Most notably, the Srivatsan model implements a variational model and uses convolutional layers. It may be necessary to define these terms: a variational model, rather than explicitly encoding an intermediate vector representation between the encoder and decoder steps, trains to produce intermediate parameters μ and σ which define a multi-dimensional Gaussian distribution, and from which data are randomly sampled before the decoding step. Proposed by Kingma et. al [7], Variational Autoencoders (VAEs) are especially useful

for generative tasks in deep learning, as they represent a large probabilistic space of outcomes. The resulting model space is also smoother and continuous, meaning any point in the space can be reasonably interpreted in the model.

Convolution, and convolutional layers—a technique which is separate from variational modeling but can be used in tandem with that technique, as in the case of the Srivatsan model—involves a moving filter called a kernel which reduces spatial dimension, improving neural network performance on position-based data like images, sound, and language. In the case of typeface style encoding, this is to dissociate style elements from their particular location in training images. For example, the serif elements of a character in a serif typeface (i.e. the tail at the ends of characters like T and S) can appear at many different locations in a training image depending on the particular character and typeface; using convolutional layers allows the model to treat these positionally-variant aspects as similar regardless of their specific location in a training image. Therefore, a model can better identify serif elements across many different characters and different serif typefaces.

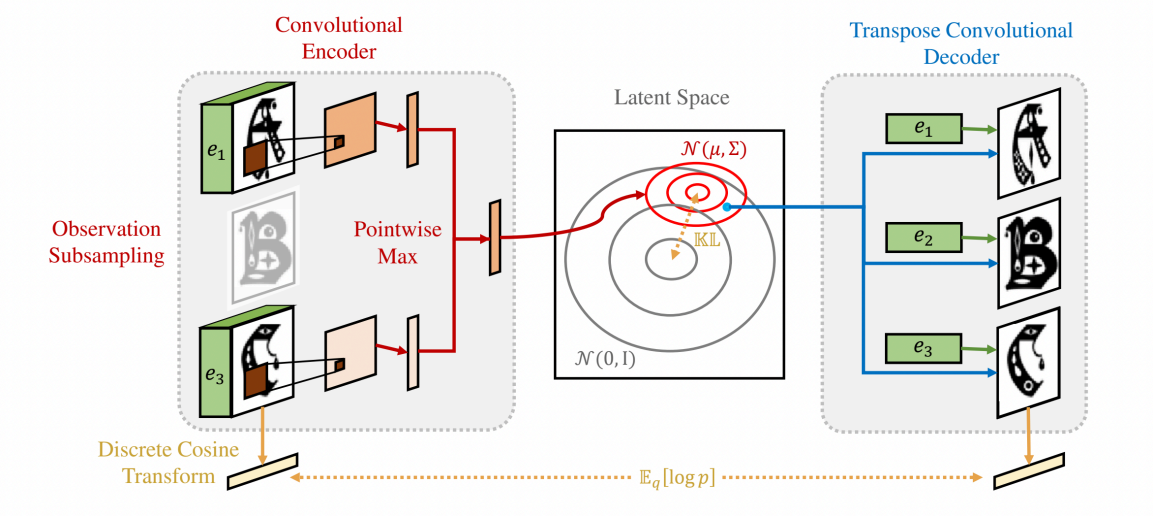

Figure 3.4: Diagram of Srivatsan et al. model architecture

Lastly, the model approaches style encoding using a slightly different task, including a full set of characters as input data for character reconstruction, and it approaches the character reconstruction task across an entire typeface at once—rather than reconstructing each character or character pair individually. Similar to our Style Transfer model detailed in 3.2.2, the Srivatsan model aims at recreating different characters across a typeface and involves vector character embeddings for those input/output characters; however, the model takes as input a full set of characters (specifically, the uppercase English letters A-Z) and randomly masks a set number of characters for reconstruction. Thus, the model reconstructs the missing characters based on the given (non-hidden) characters in the font, as well as the character embedding information for those hidden characters. In the encoding step, the model creates a latent, Gaussian style encoding from the input font image set, and the decoding step of the model uses that latent encoding along with the respective respective character embedding for the masked glyphs to reconstruct the full character set for a given typeface.

The model also implements 2-Dimensional Discrete Cosine Transform (2-D DCT-II), both within the model and to compute the loss of the model reconstructions, to encourage the model to generate sharper images. DCT-II is a volume-preserving technique and is simply a rotation in vector space, meaning the resulting probality measurements and vector distances are preserved. A diagram of the Srivatsan model architecture can be found in Figure 3.4.

Srivatsan et al. does an adequate job at glyph reconstruction—one of the focuses of the paper—but it excels especially at generating style encodings. Figure 3.5 shows a t-SNE plot of the latent style encodings generated by the Srivatsan model, colored by both weight and Google Font style category. As the figure shows, it is quite effective at encoding both the weight of a font (bolder or lighter) and the more ambiguous aspects of its style, shown in the coarse style categories from Google Fonts.

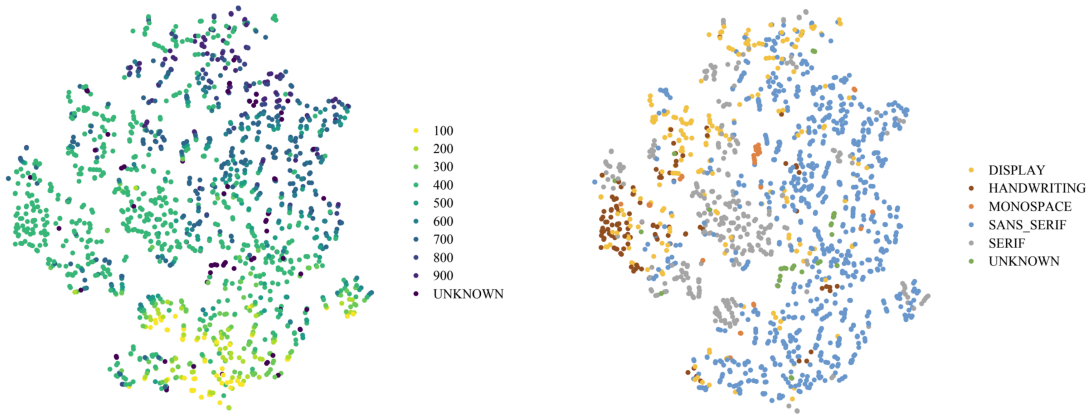

Figure 3.5: t-SNE plot of style encodings from Srivatsan et al. colored by weight (left) and Google Font style category (right)

We ported the original code from Srivatsan et al. from Python 2.7 and PyTorch 1.1.0 to Python 3.13 and PyTorch 2.5.1 and trained it on our extended Capitals64+ dataset, with 10 hidden characters per font, for 184 epochs. After training, we ran the model on a modified version of our Google Fonts dataset in order to generate useful style encodings which we could compare with our other model and use in our web app font selector tool. As with our other models, we trained the modified Srivatsan model until the MSE loss plateaued and the model seemed to adequately reconstruct the missing characters for a most input character sets.

Figure 3.6 shows 44 reconstructed character sets from the Srivatsan model at the end of our training sequence, with 10 out of 26 characters reconstructed by the model. As the figure shows, the model was better at reconstructing some fonts than others—specifically, it seems to have performed better on typefaces with more common style, such as basic serif or sans-serif fonts. Stranger, more uncommon font styles were not reconstructed as accurately; however, the character reconstruction task is only a useful metric in tracking the model's performance. Our main goal—to encode typeface style within the model—does not correspond exactly with the model's character reconstruction ability. In fact, the very accurate reconstruction performance we see for many of the typefaces,

along with a somewhat-decent reconstruction of the more difficult fonts, suggests a fairly strong representation of typeface style within the model. Further evaluations in Chapter 4 will more accurately and quantitatively analyze the style-encoding capabilities of our three models.

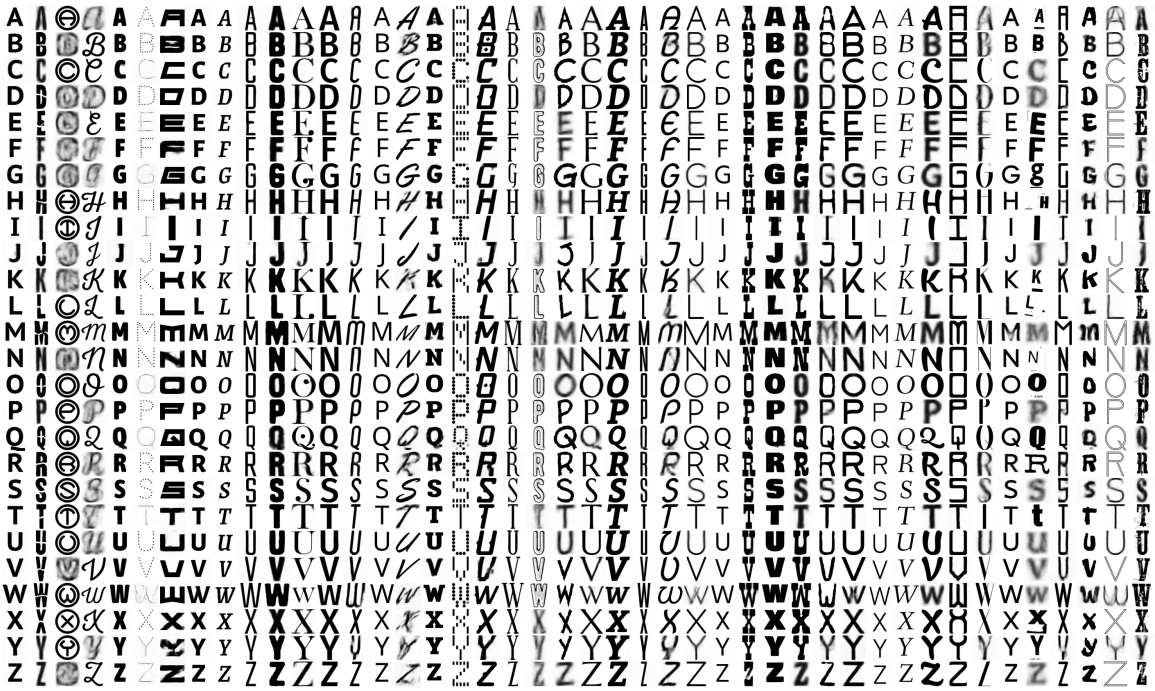

Figure 3.6: Reconstructed character sets from the adapted Srivatsan model

3.2.4 Extracting Style Encodings

In order to build useful typeface selection tools based on these models, it was necessary in all three cases to extract the model's internal style encoding vector for each typeface. This task was relatively trivial for the first two models: we simply passed each of the typeface character sets through the encoder portion of our model and performed an elementwise average across the resulting vector representation for each character, which we believed would generate a roughly-representative style encoding for the entire typeface. For the Srivatsan model, we similarly ran our model on each of the typeface character sets; however, since the Srivatsan model applies a variational approach, it was additionally necessary to perform a random sample from the Gaussian distribution defined by each typeface character set to obtain a style encoding. Because the architecture of the Srivatsan model does not train on individual character sets or pairs, but entire typeface character sets together, it was not necessary to take an average of multiple character style encodings; the model, by design, trains one generalized latent style encoding for each typeface.

3.3 System Design

In this section, we detail the process and design of turning our style encodings—generated by the models in the previous section—into a user tool for typeface selection. We consider questions of encoding dimensionality, end-to-end system design, and user experience in order to create a novel, useful typeface selector tool.

3.3.1 Style Encoding Dimensionality

One initial question which emerges when training autoencoder-like models, especially with the goal of extracting intermediate encoding vectors, is: what should be the dimensionality of those encodings? In our case especially, there is a tradeoff between encoding more information (higher dimensional vectors should be able to represent greater stylistic detail, to a certain point) and usability (lower dimensional vectors represent fewer choices for users, presumably yielding easier-to-use tools). One could, for example, train a model to generate 2-dimensional style encodings; this could create a very useful tool—we could represent this as a scatterplot of typefaces or as two sliders corresponding to the two style axes—but two dimensions is not very much space to represent the many diverse axes of typeface style. Alternatively, one could choose a very high dimensional style encoding—say, a 100-dimensional vector—which would have a much greater capacity to encode the many axes of font style; however, presenting a user with a decision for each of those dimensions would be an unwieldly and overwhelming experience for most.

We tested several embedding dimensions, but ultimately chose to encode typeface style in 6-dimensional vectors across our three models. This provides significant room for spatial exploration of typeface style, but also reasonably limits user choice, creating a better user experience. In the following section, we will describe how these six dimensions translate to a novel font selection tool; additionally, we compress these 6-dimensional style encodings into 2-dimensional space using t-SNE reduction to create an auxilliary scatterplot tool which provides users with an additional, more familiar graphical representation of the data.

3.3.2 User Experience

We experimented with several iterations of a tool which would allow users to interact with higih-dimensional style encodings in a useful manner. As mentioned previously, the question of dimensionality is a significant one when generating style encodings for user interaction; another important consideration is *how* users should interact with these high-dimensional vector spaces. Should users have full control over each dimension, or should they be guided in their decisions? How should high-dimensional space—which cannot be easily visualized or conceptualized by users—be represented? What additional tools should be provided to users (a back button, the ability to save a typeface for later reconsideration, a search bar) would help a user navigate this high-dimensional stylistic vector space? Other relevant user design questions are: What is the goal of our user (finding a specific font, finding similar fonts to a given font, or open exploration)? How much time is the user willing to spend searching for a font? What does the user want to do with this font (or fonts) once identified,

and how can we assist the user to accomplish that?

3.3.3 Lessons from Early User Tool Implementation

Early in our research, we implemented a user tool which included a numerical slider for each dimension of the model space and allowed users to generate new characters based on the model—namely, by inputting the user-determined vector into the model decoder and displaying the output generated character. Our goal was to visualize the dimensions of the model space, to see what sort of information about the characters were being encoded by the model. There were a few issues with our approach: first of all, the interface was inherently a generative task—users were generating new characters/typefaces rather than selecting a preexisting typeface in a dataset—which is a sort of task we later gave up (mostly because the resulting typefaces just don't look very good); secondly, this earlier interface was built atop our Autoencoder model, meaning the different dimensions of dictated by the sliders would change both style and character at once. Our most important takeaway, however, was that the tool afforded users too much control and not enough guidance while exploring fonts. Even with a relatively low-dimensional space like ours, giving users direct control over several continuous sliders did not make for a very good user tool. Our current interface, while still providing many different methods and dimensions of user control, is a bit more limited and guided with its options, hopefully making for a more useable tool.

Include a picture of this early implementation?

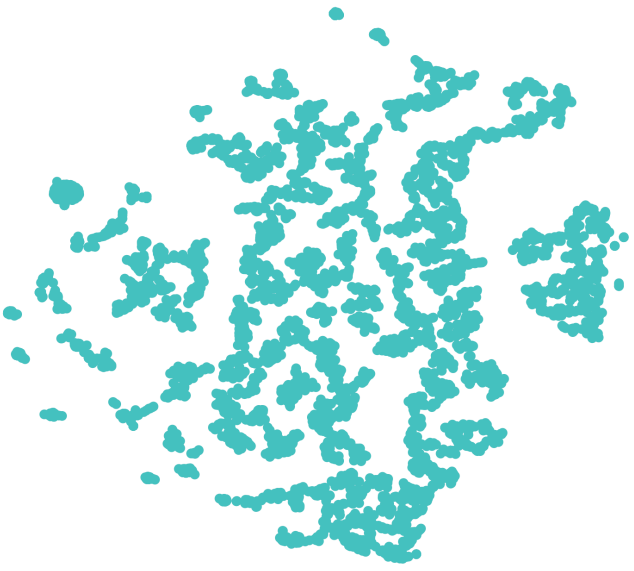

Figure 3.7: An early t-SNE scatterplot of style encodings from our model, allowing users to visually navigate a reduced-dimensionality version of typeface style encodings

3.4 Current Interface

The current iteration of our typeface selector tool involves two connected user tools which allow users to explore our 6-dimensional vector style space in two complementary ways. The first tool is an interactive scatter plot (see Figure 3.7), displaying a t-SNE reduction of our 6-dimensional style space. t-SNE is a dimensionality reduction technique which preserves local structure and clustering, which means that typefaces close to each other in the original 6-dimensional space generally remain near to each other in the reduced 2-dimensional space. The scatterplot is an intuitive and familiar representation of spatial data to most users, making it a useful tool for navigating this style-embedding space. Users can explore the scatterplot space and visualize the fonts represented at each point; if a user identifies a font that is similar to the font they are searching for or imagining, they should be able to find other, even more similar fonts by exploring the points nearby.

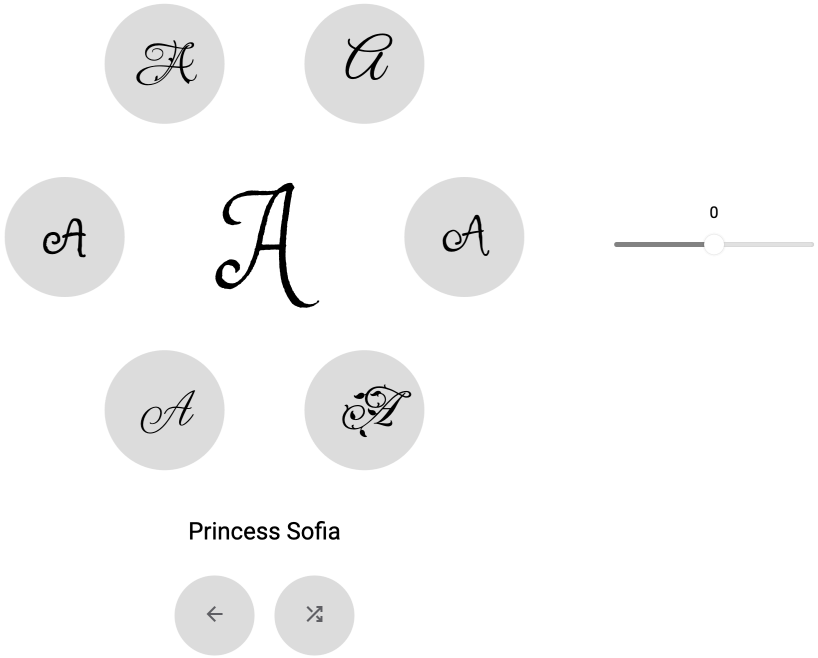

Figure 3.8: Our novel 6-dimensional typeface selector tool

In order to provide another way for users to interact with this style space—one which preserves the dimensionality of our style encodings and therefore allows users full range over the model space—we propose another typeface selector tool based on the 6-dimensional structure of our encoding data. This tool, shown in Figure 3.8, displays a center font glyph (A is the default character, but this can be changed by the user) of a randomly-selected typeface in the model space, and shows the six nearest neighbors of that font (determined by Euclidean distance) in a circle surrounding it. The slider, on the right hand side, represents magnitude; when the magnitude is zero, the six surrounding fonts represent the nearest neighbors to the center font; when the magnitude is nonzero, the tool

will search—along all six dimensions of the model space—according to the distance defined by the slider magnitude, and display the closest font in each of those dimensions. Therefore, as the slider grows further from zero, the six fonts displayed will have increasingly different style from the center font. At any point, a user can select one of the six surrounding fonts and move to that point in the model space, at which point the slider resets to zero (nearest neighbor), and the user may continue the process again in order to find a more optimal font. The tool also includes a shuffle button, enabling the uer to randomly select a new font, and a back button which allows the user to return to previously seen fonts. By providing easy-to-use buttons and dynamically displaying fonts, this tool enables users to navigate a high-dimensional model space which is difficult to conceptualize intuitively.

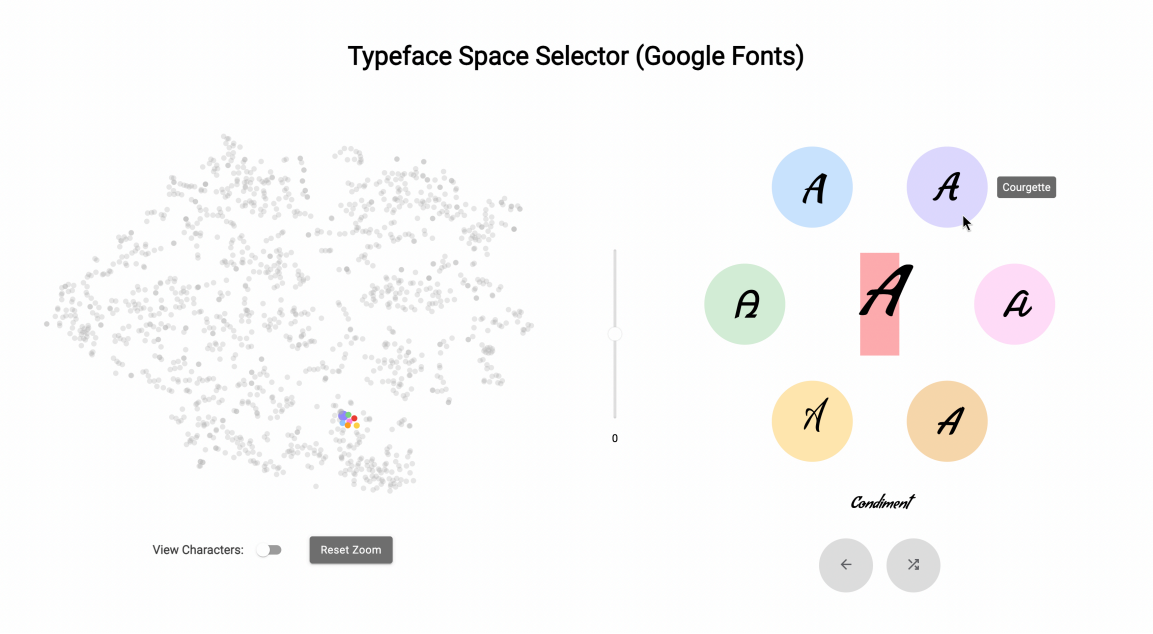

Figure 3.9: Both of our typeface selector tools side-by-side, with additional features to ease use of both tools in tandem

Our final product combines both the scatterplot and six-point tool, with several additional features to make the tools work together effectively (see Figure 3.9). Namely, we use distinct colors to display the location of each of the typefaces from the 6-point selector tool within the scatterplot visualization. Additionally, when the cursor hovers over one of the typefaces in the 6-point tool, that point grows bigger in the scatterplot tool, making it even easier and clearer to locate a typeface within the t-SNE scatterplot space. Users can click on points in both the scatterplot and the 6-point selector, and the entire tool will dynamically update to the new typeface location. Finally, there is a toggle under the scatterplot to display characters instead of circular points (maintaining colors) which makes for an easier visualization of the entire space, and users can also zoom and pan in the scatterplot tool to better navigate and explore different areas of the scatterplot (see Figure 3.10). Because of the relatively-slow rendering of these fonts, however, displaying characters on the

scatterplot does somewhat slow down the zoom capability of the scatterplot.

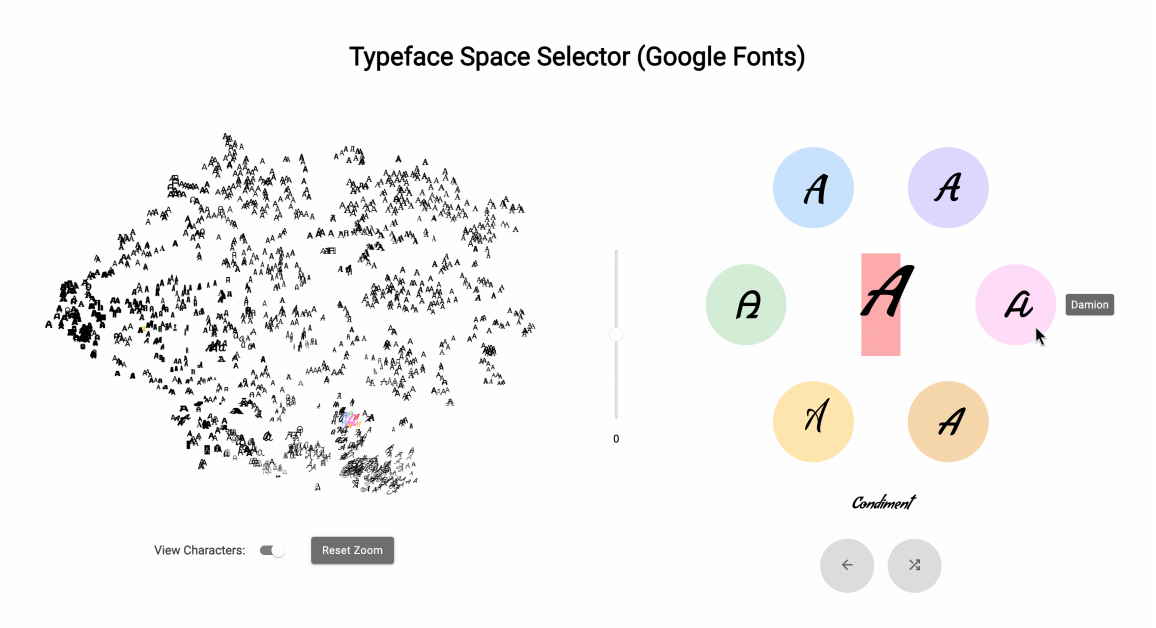

Figure 3.10: Our typeface selector tool with characters displayed on scatterplot

In order to make our implementation more streamlined, this font selector is currently limited to the Google Fonts collection of typefaces. (This choice is described further in 3.4.2.) It would not be too difficult to expand support for fonts outside of the Google Fonts library, however it might require hosting SVG files of the font characters instead of loading full font files. (Consequently, this would likely speed up the tool's performance.) The current implementation of our font selector tool can be found here.[3]

3.4.1 Backend (Flask)

We implement the backend of our web app in Flask,[4] a lightweight Python web server which adds expands capability on top of the basic HTTP GET and POST requests and includes support for URL parameters. The backend server provides two functionalities: serving t-SNE data and performing nearest-neighbor calculations. In the first case, the backend serves the full t-SNE dataset for use in our scatterplot selector tool (a small file <1MB) upon request from the frontend web app. As its second function, the server can be queried to navigate the 6-dimensional model space and compute the nearest neighbor calculations necessary for the six-point font selector tool. This backend computation—built on the Facebook AI Similarity Search (FAISS) library,[5] which provides efficient, high-dimensional vector similarity search—is as follows: given an input typeface and magnitude, the server first locates the style encoding for the input typeface, then creates six new style encoding

[3] http://sysnet.cs.williams.edu/~25sm39/
[4] https://flask.palletsprojects.com/en/stable/
[5] https://ai.meta.com/tools/faiss/

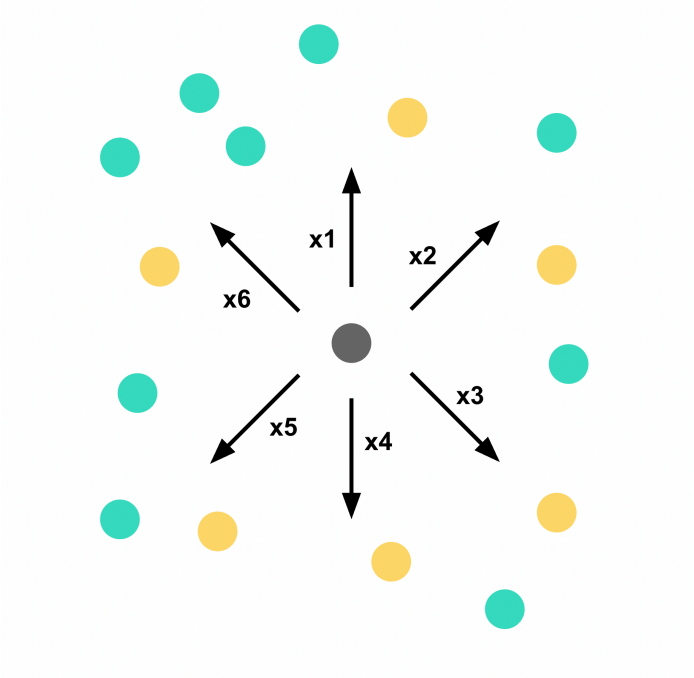

Figure 3.11: Search algorithm in our model space: extend search in each dimension according to magnitude and find nearest neighbor

vectors with the magnitude value added in each of the corresponding dimensions (see Figure 3.11). FAISS then finds the nearest actual typeface style encoding for each of the six calculated vectors (avoiding duplicates when possible) and serves those font names to the client in a JSON file.

3.4.2 Frontend (React)

Our frontend is implemented in React,[6] an open-source JavaScript library built to create interactive web applications using modular components. The scatterplot tool uses the Chart.js library[7] for simple, efficient plot creation, and the 6-point font selector was built by hand using React components. For rendering the actual fonts, our frontend uses the GoogleFontLoader package for React,[8] which facilitates easy, dynamic font loading on webpages. This allows us to avoid serving the 2.7GB of font binary files, alleviating significant server load; however, it limits our current implementation to typefaces from the Google Fonts library. This is not necessarily a bad thing—Google Fonts is widely-used and contains a wide range of different fonts—but it does mean that common proprietary fonts such as Times New Roman and Comic Sans are also not included in the current version of our font selector tool. A diagram of our system architecture can be found in Figure 3.12.

[6] https://react.dev/
[7] https://www.chartjs.org/
[8] https://www.npmjs.com/package/react-google-font-loader

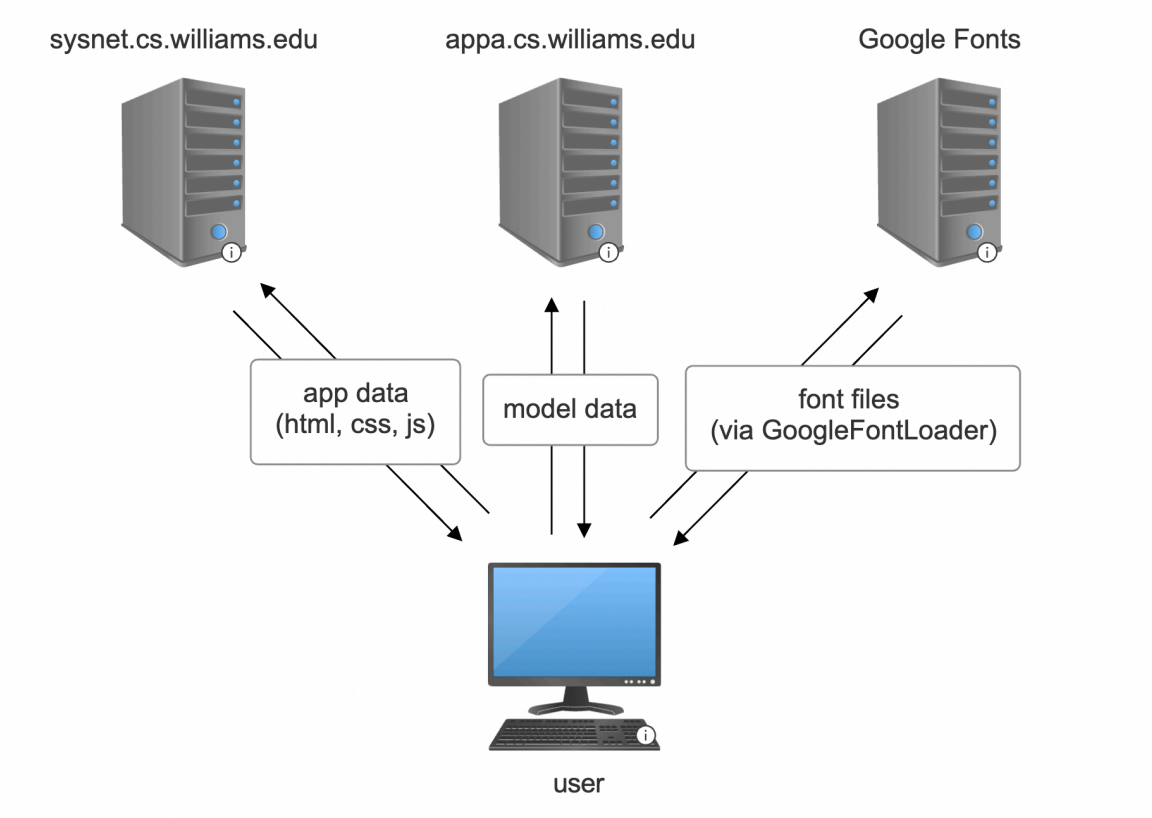

Figure 3.12: A network diagram of our font selector web app system

Chapter 4

Evaluation

In this chapter we evaluate the performance of our model and novel font selection interface. Our Model Evaluation section includes a quantiative analysis of our model encodings, calulating the average pairwise distances of font groups in our model space according to the novel Google Fonts typeface categories discussed in Section 2.1.2. In our User Evaluation section, on the other hand, we practically test our font selection interface against two alternate font selection tools with a small user study involving a font matching task and an open-ended evaluation section. While the user evaluation certainly evaluates the usability and effectiveness of our selection interface, the model space itself—how accurately the model encodes typeface style—certainly affects the usability of the tool as well. Therefore, this user evaluation quantifies the combined effectiveness of the model encodings and the selector interface together.

4.1 Model Evaluation

We conduct our model evaluation by measuring the average pairwise distances between fonts within the new typeface style categories defined by the Google Fonts library (see Section 2.1.2). While these categories are not perfect in their style accuracy (as previously discussed), they provide a useful font-grouping metric which is entirely separate from our model training, and therefore could serve as a useful model evaluation tool.

Table 4.1: Average pairwise Euclidean distance between style encodings grouped by Google Fonts categories. Abbreviated from Table A.1 (Appendix).

Category	**Autoencoder**	**Style Transfer**	**Srivatsan C64**	**Srivatsan Full**
average category distance	1.318	1.337	**0.952**	**0.769**
appearance-art-deco	1.728	1.337	1.184	**0.780**
appearance-art-nouveau	**0.453**	**0.737**	**0.783**	**0.706**
appearance-blackletter	**0.698**	**0.956**	**0.956**	**0.921**
appearance-blobby	1.198	1.276	**0.880**	**0.657**

Continued on next page

Table 4.1 continued from previous page

Category	Autoencoder	Style Transfer	Srivatsan C64	Srivatsan Full
appearance-distressed	1.489	1.278	1.057	**0.698**
appearance-inline	2.238	1.832	1.209	**0.732**
...				
serif-modern	**0.794**	**0.596**	**0.847**	**0.889**
serif-old-style	**0.522**	**0.753**	**0.695**	**0.814**
serif-slab	1.582	1.347	1.117	**0.845**
serif-transitional	**0.551**	**0.667**	**0.732**	**0.954**

Table 4.1 displays some of the average pairwise Euclidean distance data from this evaluation. Each row contains data for a different Google Fonts category, and the four columns represent our four models. ("Srivatsan C64" is an earlier version of the Srivatsan model, trained only on the Capitals64 dataset, while "Srivatsan Full" is the Srivatsan model trained on our full dataset.) We find that the Srivatsan model outperforms both the Autoencoder and Style Transfer models, with a greater number of Google Fonts categories having a lower average distance between fonts than the overall pairwise average, and the average distance score across all the categories slightly lower than the overall average (0.952). However, when trained on our full dataset, the Srivatsan model performs even better, with all but one of the Google Font categories having a closer-than-average distance score, and an average distance score across all the categories of 0.769.

It makes sense that the Srivatsan model would perform better with our full dataset: partially, this may be due to simply having a larger training dataset; but this is also likely a result of having seen the Google Fonts in its training. In the latter instance, the Srivatsan model was able to actively optimize the style encodings for the Google Fonts typefaces, while the more limited Srivatsan model did not have the opportunity to specifically optimize these style encodings. This does not, however, mean that the Google Fonts style encodings of the full Srivatsan model are of lower quality; rather, having the ability to specifically train on these fonts should result in better, more accurate style encodings for use in our font selection tool.

Looking at the extended data in Table A.1 (Appendix), we can also make some conclusions about the different models' ability to encode certain types of style as designated by the Google Fonts categories. For example, we find that all model implementations encode sans-serif style quite effectively, as the average distance between most of the sans-serif style groups have a lower average pairwise distance than the overall average distance in their model spaces. The same is true for most of the serif style groups, however only the fully trained Srivatsan model seems capable of encoding the style of the serif-fatface and serif-slab groups. This makes some sense, as these two categories contain less "typical" serif style fonts (see Figures 4.1 and 4.2), but the groups are also relatively small (15 and 68 fonts, respectively) and it is hard to make a definite conclusion with such a small sample size.

We additionally find that some of the more ambiguous font style categories are not well-encoded by our simpler models. In particular, categories in the appearance, feeling, and seasonal style groups

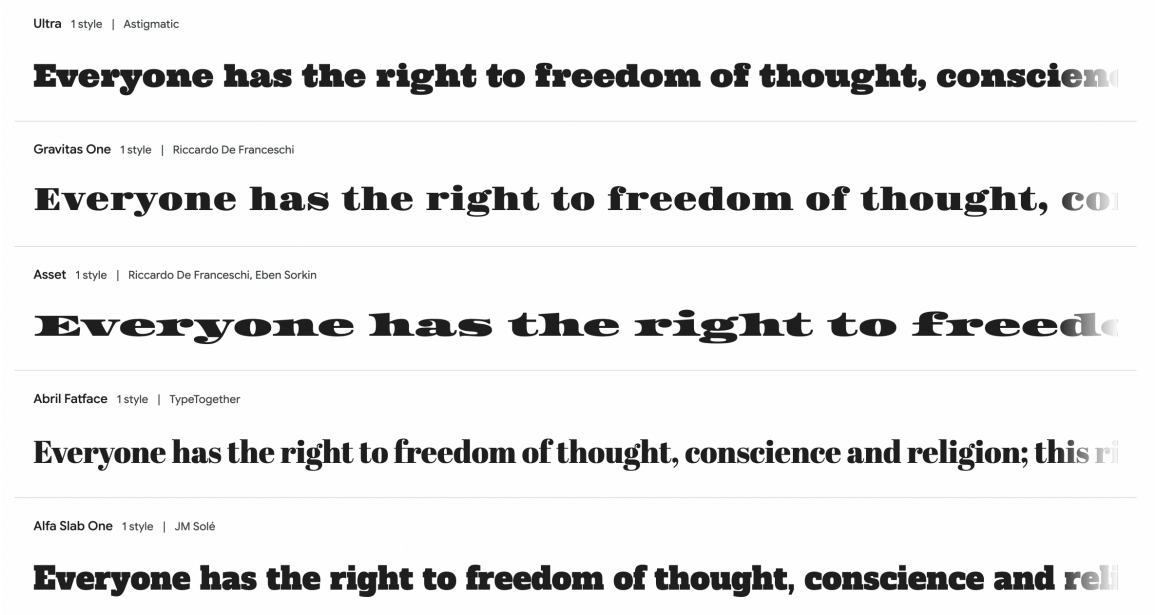

Figure 4.1: Selection of fonts in the Google Fonts serif-fatface category

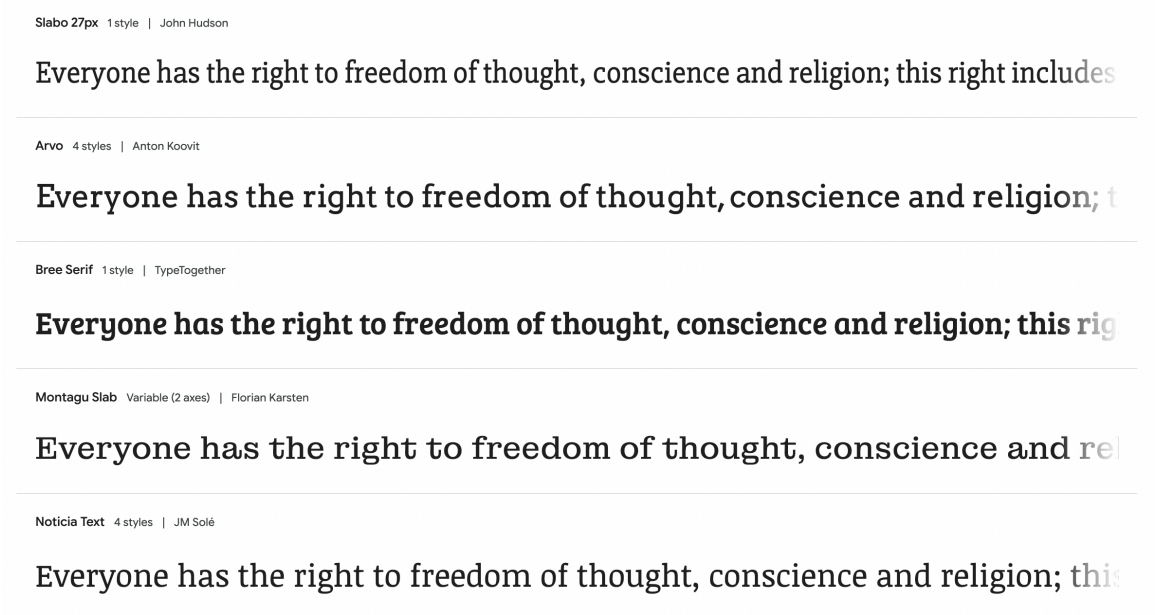

Figure 4.2: Selection of fonts in the Google Fonts serif-slab category

Figure 4.3: Selection of fonts in the Google Fonts feeling-artistic category

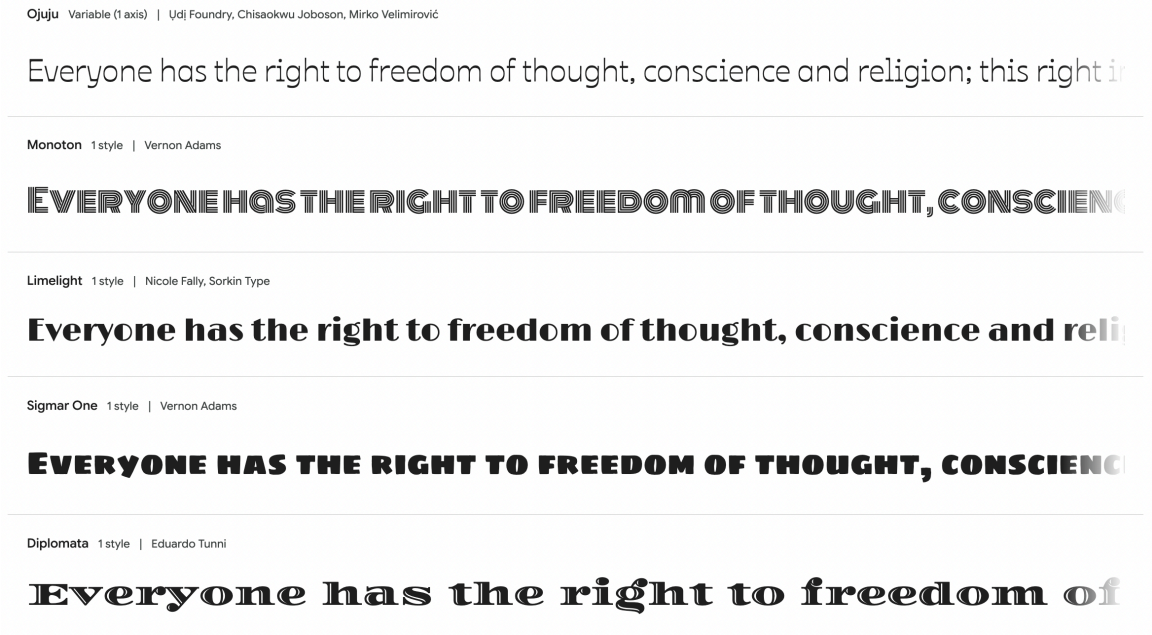

Figure 4.4: Selection of fonts in the Google Fonts seasonal-kwanzaa category

receive poor similarity scores in the stylistic encoding space generated by the more basic models. For many of these ambiguous style categories, only the Srivatsan model trained on our full dataset seemed to represent the typeface style group effectively. Many of these ambiguous stylistic categories contain a wide variety of different font styles, making it much more difficult to quantify their style. For example, the fonts in the feeling-artistic and seasonal-kwanzaa categories (see Figures 4.3 and 4.4) do not seem to follow one particular or obvious style. However, it is notable and impressive that our final model is capable of representing these more abstract style groups, and it suggests that these model encodings should provide a strong foundation for style-based font selection.

4.1.1 A Note: Euclidean Distance vs Cosine Similarity

We find an interesting result when comparing the Cosine and Euclidean distances in our model evaluation. In general, we find that the models which perform strongly for Euclidean distance do not perform as well when considering Cosine similarity; and conversely, the models which perform less well in terms of Euclidean closeness perform better with Cosine similarity. For example, when comparing the normalized category-wise Euclidean distances and Cosine similarities of the Autoencoder model (see Table 4.2), we find that a greater number of categories have a better-than-average similarity when considering Cosine similarity rather than Euclidean distance; and the average similarity score across all category groups is closer than average (1.125) when using Cosine similarity but further than average (1.318) for Euclidean distance. On the other hand, our full Srivatsan model performs very well on Euclidean distance scores, but not as well when using Cosine similarity (see Table 4.3), with the average category-wise Cosine similarity score (0.987) slightly worse than the overall pairwise average for the full model space. It is unclear exactly why there is a discrepancy with Cosine and Euclidean similarity evaluation in our model space—it is certainly the case that Euclidean and Cosine similarity metrics are measuring different types of vector similarity, which might be representing different aspects of the model encodings—but for the purposes of our selection tool this should not matter: since our user tool is built around Euclidean distance metrics, it is mainly important that our full Srivatsan model performs well on Euclidean distance metrics—that similar font styles should be closer to each other in Euclidean space—which it does. Cosine similarity metrics tell an interesting story about our model space, but ultimately the metric of most importance for our purposes is Euclidean distance.

Table 4.2: Normalized average pairwise Euclidean and Cosine distances across Google Fonts category groups in our Autoencoder model space. Abbreviated from Table A.2 (Appendix).

Category	**Autoencoder (Euclidean)**	**Autoencoder (Cosine)**
average category distance	1.318	**1.125**
appearance-art-deco	1.728	**1.282**
appearance-art-nouveau	**0.453**	0.494
appearance-blackletter	**0.698**	**1.101**
appearance-blobby	1.198	0.906

Continued on next page

Table 4.2 continued from previous page

Category	Autoencoder (Euclidean)	Autoencoder (Cosine)
appearance-distressed	1.489	**1.434**
appearance-inline	2.238	**1.692**
appearance-lunar-new-year	1.390	**1.742**
appearance-marker	1.139	**1.045**
appearance-medieval	**0.934**	0.848
appearance-monospaced	**0.475**	0.866
appearance-not-text	6.477	**2.699**
...		
serif-didone	1.238	0.853
serif-fatface	2.478	0.666
serif-humanist	**0.628**	0.535
serif-modern	**0.794**	0.955
serif-old-style	**0.522**	0.544
serif-scotch	**0.562**	0.808
serif-slab	1.582	0.920
serif-transitional	0.551	0.529

Table 4.3: Normalized average pairwise Euclidean and Cosine distances across Google Fonts category groups in our full Srivatsan model space. Abbreviated from Table A.3 (Appendix).

Category	Srivatsan Full (Euclidean)	Srivatsan Full (Cosine)
average category distance	**0.769**	0.987
appearance-art-deco	**0.780**	**1.266**
appearance-art-nouveau	**0.706**	0.650
appearance-blackletter	**0.921**	0.881
appearance-blobby	**0.657**	0.955
appearance-distressed	**0.698**	**1.303**
appearance-inline	**0.732**	**1.140**
appearance-lunar-new-year	**0.841**	**1.264**
appearance-marker	**0.651**	**1.104**
appearance-medieval	**0.758**	0.855
appearance-monospaced	**0.746**	0.909
appearance-not-text	**0.754**	**1.707**
...		
serif-humanist	**0.841**	0.525
serif-modern	**0.889**	0.597
serif-old-style	**0.814**	0.477

Continued on next page

Table 4.3 continued from previous page

Category	**Srivatsan Full (Euclidean)**	**Srivatsan Full (Cosine)**
serif-scotch	**0.891**	0.502
serif-slab	**0.845**	0.849
serif-transitional	**0.954**	0.468

4.2 User Evaluation

This section will be written once the user studies are complete.

Chapter 5

Conclusion

In this thesis, we have detailed our research into the potential use of Autoencoder-like neural networks to encode typeface style. The most common font selection tools largely ignore style as an aspect of typeface selection, making it difficult or impossible to ask questions like "Which fonts are most similar to Futura?" or "What is a font which is similar to Times New Roman but more playful?" The first half of our research involved gathering a large dataset of font character data representative of a wide range of typeface styles, including character sets from 14,391 typefaces across the Google Fonts and Apple libraries and the Capitals64 dataset (see Section 3.1), and training several models based on the original Autoencoder model to encode typeface style vectors of input typeface character sets. The second half of this project involved building a useful user tool on these typeface style encoding data, in orde to demonstrate our hypothesis that the style encodings generated by these Autoencoder-based neural networks can serve as a useful foundation for style-based font selection tools. Finally, we evaluate our model and our novel font selection tool based on both quantitative distance measurements, as well as quantitative and qualitative data from a small user study.

5.1 Findings

We find that many of our models succeed in encoding certain aspects typeface style, but our model adapted from Srivatsan et al. [11] and trained on our larger dataset performed the strongest when evaluated against the novel font attribute categories created by the Google Fonts library: when looking at Euclidean distance as a metric of style vector similarity, all but one of the Google Fonts style categories corresponded to a closer-than-average Euclidean similarity score. The other models did not capture as many of these categories, and we generally find that models which were more complex and disentangled character (A or B, e.g.) from style (determined by typeface) performed better on these Euclidean similarity metrics. In general, "easier" style categories—such as serif and sans-serif category groups—were more likely to have a closer-than-average similarity score across the models, while more abstract and diverse style categories like feeling-loud or seasonal-kwanzaa were less likely to have closer-than-average distance scores. We also measured these similarity scores using Cosine similarity distances, and found that these scores told a somewhat different story than the

Euclidean distance metrics—the models which performed better under Euclidean distance similarity tended to perform poorer under Cosine similarity, and vise-versa—but because our font selection tool was built on Euclidean distance similarity, we determine that these differing Cosine similarity results are not too important for our purposes.

Write about user study findings

5.2 Future Work

There are many aspects of this research which could be improved upon or investigated further. This section details those areas of future work, split into two overall categories: model-based improvements, and interface-based improvements.

5.2.1 Model Improvements

There are a large diversity of approaches to building neural image models, and for that reason much time could be spent implementing different models on font image data and evaluating which models perform best. The scope of this thesis research allowed only enough time to implement and evaluate a few models (Basic Autoencoder, Style Transfer, and the Srivatsan et al. model), but it would be interesting and fruitful to explore a wider range of model approaches. However, I believe there is a great potential in style encoding models for typeface based not on bitmap pixel images, but on vector graphics which encode geometric shapes rather than pixel values. In fact, this is the native representation of font files—which ensures that fonts can be viewed clearly and without pixelation at any scale—and I think that building font reconstruction and style encoding models based on this non-pixel representation could potentially yield more effective style representations. This would also yield much cleaner reconstructed fonts (whose representation would match the native method of representing fonts), making generative tasks (i.e. creating *new* fonts based on existing data) much more realistic. Currently, the font images generated by bitmap-based models look (at best) fuzzy and pixelated, far from an actual useable font. Certain groups such as Carlier et al. [3] have already begun to explore these directions in font generation, but certainly much more work can be done to explore this area of font style encoding and generation.

5.2.2 Interface Improvements

It is fair to say that our final font selection interface, while user friendly in certain ways, provides users with a bit too much control over the model space. While this is okay for our proof-of-concept—in order to demonstrate our hypothesis that these model style encodings could be used to create a useful style-based font selection tool—there is certainly a lot of room for improvement in the tool's interface. Future work, especially if this interface were to be built into a production-grade tool, would likely involve simplifying the interface to make it more approachable. Additionally, while our current implementation only includes typefaces from the Google Fonts library for the purposes of simplification, it would certainly be possible to support a wider range of fonts; however, it would

probably be necessary to adapt the interface to use SVG character files rather than loading whole font files. Given the current implementation, at a certain point with enough typefaces loaded at once, the website interface would likely become unusably slow.

5.3 Lessons Learned

The process of this thesis research, spanning eight months of my undergraduate career, has been a significant undertaking. I have grown as a programmer, a researcher, and a student—and there are, admittedly, things I would have done differently looking back. For one, I was less diligent than I would have liked with the organization of my code. In my thesis directory alone I have written over 130 Python and Bash scripts, many of which should have been combined and condensed. Additionally, I did not document them as well as I could have; I regret this. However, if you are reading this, there is final codebase where I have uploaded the important scripts to reproduce this project, and in those files I have attempted to make my work as clear and well-documented as possible.

I have learned that research—especially with good advisors, family, and friends supporting you—is an incredibly rewarding process. It is an experience which teaches you much more about yourself, your approaches to work, your motivation and drive, and how to commit to an endeavor to the very end. For this experience, I am incredibly grateful to all those aforementioned people who have helped me in working towards this final product. Now, it is time for a nice, long nap.

Bibliography

[1] Azadi, S., Fisher, M., Kim, V., Wang, Z., Shechtman, E., and Darrell, T. Multi-Content GAN for Few-Shot Font Style Transfer, 2017.

[2] Bank, D., Koenigstein, N., and Giryes, R. Autoencoders, 2021.

[3] Carlier, A., Danelljan, M., Alahi, A., and Timofte, R. Deepsvg: A hierarchical generative network for vector graphics animation. In *Advances in Neural Information Processing Systems* (2020), vol. 33, pp. 2262–2273.

[4] Cheng, K. *Designing Type*. Yale University Press, New Haven, CT, 2006.

[5] Cho, J., Lee, K., and Choi, J. Y. Font Representation Learning via Paired-glyph Matching, 2022.

[6] Kingma, D. P., and Welling, M. Auto-Encoding Variational Bayes, 2013.

[7] Kingma, D. P., and Welling, M. An Introduction to Variational Autoencoders. *Foundations and Trends in Machine Learning 12*, 4 (2019), 307–392.

[8] O'Donovan, P., Lieberman, E., Zhao, K., Fry, E., Isenberg, P., and Hertzmann, A. Exploratory Font Selection Using Crowdsourced Attributes. In *Proceedings of the 27th annual ACM symposium on User interface software and technology* (2014), ACM, pp. 369–378.

[9] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. *Learning Internal Representations by Error Propagation*. MIT Press, Cambridge, MA, USA, 1986, p. 318–362.

[10] Shaikh, A. D., Chaparro, B. S., and Fox, D. Perception of fonts: Perceived personality traits and uses. *Usability News 8*, 1 (February 2006).

[11] Srivatsan, N., Barron, J. T., Klein, D., and Berg-Kirkpatrick, T. A Deep Factorization of Style and Structure in Fonts, 2020.

Appendix

Table A.1: Average pairwise Euclidean distance between style encodings grouped by Google Fonts categories, with distances normalized relative to the average pairwise distance between all fonts in model space. Distances less than one (categories whose average distance is less than the overall pairwise distance of the model) are in bold. The average of all category-wise distance scores is shown at top.

Category	**Autoencoder**	**Style Transfer**	**Srivatsan C64**	**Srivatsan Full**
average category distance	1.318	1.337	**0.952**	**0.769**
appearance-art-deco	1.728	1.337	1.184	**0.780**
appearance-art-nouveau	**0.453**	**0.737**	**0.783**	**0.706**
appearance-blackletter	**0.698**	**0.956**	**0.956**	**0.921**
appearance-blobby	1.198	1.276	**0.880**	**0.657**
appearance-distressed	1.489	1.278	1.057	**0.698**
appearance-inline	2.238	1.832	1.209	**0.732**
appearance-lunar-new-year	1.390	1.541	1.093	**0.841**
appearance-marker	1.139	1.208	**0.870**	**0.651**
appearance-medieval	**0.934**	1.012	**0.924**	**0.758**
appearance-monospaced	**0.475**	**0.616**	**0.717**	**0.746**
appearance-not-text	6.477	11.048	**0.863**	**0.754**
appearance-pixel	1.424	1.310	**0.917**	**0.436**
appearance-shaded	2.713	1.802	1.057	**0.649**
appearance-stencil	1.227	1.213	1.117	**0.842**
appearance-techno	1.812	1.408	1.087	**0.891**
appearance-tuscan	2.130	1.238	1.046	**0.633**
appearance-valentines	1.861	1.671	1.072	**0.729**
appearance-wacky	1.674	1.244	1.086	**0.722**
appearance-wood-type	2.481	1.904	1.159	**0.752**
calligraphy-all	1.119	1.358	**0.862**	**0.615**
calligraphy-formal	**0.967**	1.237	**0.622**	**0.406**
calligraphy-handwritten	1.040	1.241	**0.827**	**0.624**

Continued on next page

Table A.1 continued from previous page

Category	Autoencoder	Style Transfer	Srivatsan C64	Srivatsan Full
calligraphy-informal	1.155	1.376	**0.856**	**0.582**
feeling-active	**0.963**	1.075	**0.882**	**0.621**
feeling-artistic	1.222	1.434	**0.896**	**0.646**
feeling-awkward	1.238	1.185	**0.992**	**0.694**
feeling-business	**0.646**	0.575	**0.907**	**0.996**
feeling-calm	**0.735**	0.662	**0.851**	**0.939**
feeling-childlike	**0.909**	1.014	**0.902**	**0.647**
feeling-cute	1.038	1.130	**0.962**	**0.736**
feeling-excited	1.390	1.257	1.043	**0.667**
feeling-fancy	1.030	1.303	**0.629**	**0.385**
feeling-futuristic	1.405	1.124	1.069	**0.983**
feeling-happy	0.932	**0.909**	**0.979**	**0.699**
feeling-innovative	2.537	1.938	1.160	**0.741**
feeling-loud	1.665	1.481	1.100	**0.805**
feeling-playful	1.546	1.341	1.079	**0.755**
feeling-rugged	1.564	1.336	1.074	**0.725**
feeling-sophisticated	1.015	1.275	**0.646**	**0.439**
feeling-stiff	1.016	**0.858**	1.041	**0.975**
feeling-vintage	1.042	1.031	1.023	**0.937**
sans-serif-all	0.828	**0.726**	**0.859**	**0.959**
sans-serif-geometric	1.100	**0.799**	**0.967**	**0.839**
sans-serif-glyphic	**0.743**	**0.671**	**0.812**	**0.823**
sans-serif-grotesque	**0.852**	**0.958**	**0.945**	**0.945**
sans-serif-humanist	**0.727**	**0.629**	**0.786**	**0.940**
sans-serif-neo-grotesque	**0.731**	**0.673**	**0.840**	**0.918**
sans-serif-rounded	**0.836**	**0.873**	**0.901**	**0.876**
sans-serif-superellipse	**0.810**	**0.945**	**0.955**	1.014
seasonal-christmas	1.440	1.637	1.106	**0.750**
seasonal-diwali	1.173	1.223	1.011	**0.823**
seasonal-halloween	1.721	1.931	1.048	**0.681**
seasonal-hanukkah	1.274	1.107	1.069	**0.892**
seasonal-kwanzaa	1.498	1.318	1.084	**0.866**
seasonal-lunar-new-year	1.174	1.446	0.978	**0.831**
seasonal-valentines	1.861	1.671	1.072	**0.729**
serif-all	**0.840**	**0.794**	**0.889**	**0.927**
serif-didone	1.238	1.363	**0.846**	**0.688**
serif-fatface	2.478	2.896	1.082	**0.755**
serif-humanist	**0.628**	**0.545**	**0.873**	**0.841**

Continued on next page

Table A.1 continued from previous page

Category	**Autoencoder**	**Style Transfer**	**Srivatsan C64**	**Srivatsan Full**
serif-modern	**0.794**	**0.596**	**0.847**	**0.889**
serif-old-style	**0.522**	**0.753**	**0.695**	**0.814**
serif-scotch	**0.562**	**0.576**	**0.879**	**0.891**
serif-slab	1.582	1.347	1.117	**0.845**
serif-transitional	**0.551**	**0.667**	**0.732**	**0.954**

Table A.2: Average pairwise Euclidean and Cosine distances between style encodings in the Autoencoder model space, across Google Fonts categories, with distances normalized relative to average pairwise distance across entire model space. Distance values smaller than the overall pairwise average (less than one for Euclidean, greater than one for Cosine) are bolded. The average of all category-wise distance scores is shown at top.

Category	**Autoencoder (Euclidean)**	**Autoencoder (Cosine)**
average category distance	1.318	**1.125**
appearance-art-deco	1.728	**1.282**
appearance-art-nouveau	**0.453**	0.494
appearance-blackletter	**0.698**	**1.101**
appearance-blobby	1.198	0.906
appearance-distressed	1.489	**1.434**
appearance-inline	2.238	**1.692**
appearance-lunar-new-year	1.390	**1.742**
appearance-marker	1.139	**1.045**
appearance-medieval	**0.934**	0.848
appearance-monospaced	**0.475**	0.866
appearance-not-text	6.477	**2.699**
appearance-pixel	1.424	0.8496
appearance-shaded	2.713	**2.086**
appearance-stencil	1.227	**1.412**
appearance-techno	1.812	**1.213**
appearance-tuscan	2.130	**1.324**
appearance-valentines	1.861	**1.466**
appearance-wacky	1.674	**1.311**
appearance-wood-type	2.481	**1.296**
calligraphy-all	1.119	**1.138**
calligraphy-formal	**0.967**	0.540
calligraphy-handwritten	1.040	**1.178**
calligraphy-informal	1.155	**1.115**
feeling-active	**0.963**	0.966

Continued on next page

Table A.2 continued from previous page

Category	Autoencoder (Euclidean)	Autoencoder (Cosine)
feeling-artistic	1.222	**1.198**
feeling-awkward	1.238	**1.270**
feeling-business	**0.646**	0.719
feeling-calm	**0.735**	0.886
feeling-childlike	**0.909**	**1.079**
feeling-cute	1.038	**1.156**
feeling-excited	1.390	**1.428**
feeling-fancy	1.030	0.582
feeling-futuristic	1.405	**1.246**
feeling-happy	**0.932**	**1.088**
feeling-innovative	2.537	**1.611**
feeling-loud	1.665	**1.157**
feeling-playful	1.546	**1.288**
feeling-rugged	1.564	**1.489**
feeling-sophisticated	1.015	0.658
feeling-stiff	1.016	0.968
feeling-vintage	1.042	**1.042**
sans-serif-all	**0.828**	0.885
sans-serif-geometric	1.100	**1.026**
sans-serif-glyphic	**0.743**	0.890
sans-serif-grotesque	**0.852**	**1.181**
sans-serif-humanist	**0.727**	0.758
sans-serif-neo-grotesque	**0.731**	0.922
sans-serif-rounded	**0.836**	**1.082**
sans-serif-superellipse	**0.810**	**1.112**
seasonal-christmas	1.440	**1.476**
seasonal-diwali	1.173	**1.144**
seasonal-halloween	1.721	**1.500**
seasonal-hanukkah	1.274	**1.406**
seasonal-kwanzaa	1.498	**1.153**
seasonal-lunar-new-year	1.174	**1.740**
seasonal-valentines	1.861	**1.466**
serif-all	**0.840**	0.715
serif-didone	1.238	0.853
serif-fatface	2.478	0.666
serif-humanist	**0.628**	0.535
serif-modern	**0.794**	0.955
serif-old-style	**0.522**	0.544

Continued on next page

Table A.2 continued from previous page

Category	**Autoencoder (Euclidean)**	**Autoencoder (Cosine)**
serif-scotch	**0.562**	0.808
serif-slab	1.582	0.920
serif-transitional	0.551	0.529

Table A.3: Average pairwise Euclidean and Cosine distances between style encodings in the full Srivatsan model space, across Google Fonts categories, with distances normalized relative to average pairwise distance across entire model space. Distance values smaller than the overall pairwise average (less than one for Euclidean, greater than one for Cosine) are bolded. The average of all category-wise distance scores is shown at top.

Category	**Srivatsan Full (Euclidean)**	**Srivatsan Full (Cosine)**
average category distance	**0.769**	0.987
appearance-art-deco	**0.780**	**1.266**
appearance-art-nouveau	**0.706**	0.650
appearance-blackletter	**0.921**	0.881
appearance-blobby	**0.657**	0.955
appearance-distressed	**0.698**	**1.303**
appearance-inline	**0.732**	**1.140**
appearance-lunar-new-year	**0.841**	**1.264**
appearance-marker	**0.651**	**1.104**
appearance-medieval	**0.758**	0.855
appearance-monospaced	**0.746**	0.909
appearance-not-text	**0.754**	**1.707**
appearance-pixel	**0.436**	**1.300**
appearance-shaded	**0.649**	**1.362**
appearance-stencil	**0.842**	**1.214**
appearance-techno	**0.891**	**1.301**
appearance-tuscan	**0.633**	**1.054**
appearance-valentines	**0.729**	**1.018**
appearance-wacky	**0.722**	**1.223**
appearance-wood-type	**0.752**	**1.199**
calligraphy-all	**0.615**	0.933
calligraphy-formal	**0.406**	0.393
calligraphy-handwritten	**0.624**	0.986
calligraphy-informal	**0.582**	0.933
feeling-active	**0.621**	0.924
feeling-artistic	**0.646**	0.873
feeling-awkward	**0.694**	**1.215**

Continued on next page

Table A.3 continued from previous page

Category	Srivatsan Full (Euclidean)	Srivatsan Full (Cosine)
feeling-business	**0.996**	0.756
feeling-calm	**0.939**	0.892
feeling-childlike	**0.647**	**1.104**
feeling-cute	**0.736**	**1.124**
feeling-excited	**0.667**	**1.301**
feeling-fancy	**0.385**	0.374
feeling-futuristic	**0.983**	**1.189**
feeling-happy	**0.699**	**1.111**
feeling-innovative	**0.741**	**1.470**
feeling-loud	**0.805**	**1.086**
feeling-playful	**0.755**	**1.214**
feeling-rugged	**0.725**	**1.336**
feeling-sophisticated	**0.439**	0.377
feeling-stiff	**0.975**	**1.019**
feeling-vintage	**0.937**	0.879
sans-serif-all	**0.959**	0.902
sans-serif-geometric	**0.839**	**1.009**
sans-serif-glyphic	**0.823**	0.906
sans-serif-grotesque	**0.945**	**1.067**
sans-serif-humanist	**0.940**	0.746
sans-serif-neo-grotesque	**0.918**	0.974
sans-serif-rounded	**0.876**	**1.031**
sans-serif-superellipse	1.014	**1.053**
seasonal-christmas	**0.750**	**1.201**
seasonal-diwali	**0.823**	**1.036**
seasonal-halloween	**0.681**	**1.264**
seasonal-hanukkah	**0.892**	**1.242**
seasonal-kwanzaa	**0.866**	**1.222**
seasonal-lunar-new-year	**0.831**	**1.207**
seasonal-valentines	**0.729**	**1.018**
serif-all	**0.927**	0.577
serif-didone	**0.688**	0.399
serif-fatface	**0.755**	0.664
serif-humanist	**0.841**	0.525
serif-modern	**0.889**	0.597
serif-old-style	**0.814**	0.477
serif-scotch	**0.891**	0.502
serif-slab	**0.845**	0.849

Continued on next page

Table A.3 continued from previous page

Category	**Srivatsan Full (Euclidean)**	**Srivatsan Full (Cosine)**
serif-transitional	**0.954**	0.468