Volume 0 (1981), Number 0 pp. 1–13 COMPUTER GRAPHICS *forum*

Learning A Stroke-Based Representation for Fonts

Elena Balashova[1], Amit H. Bermano[1], Vladimir G. Kim[2], Stephen DiVerdi[2], Aaron Hertzmann[2], Thomas Funkhouser[1]
[1]Princeton University [2]Adobe Research

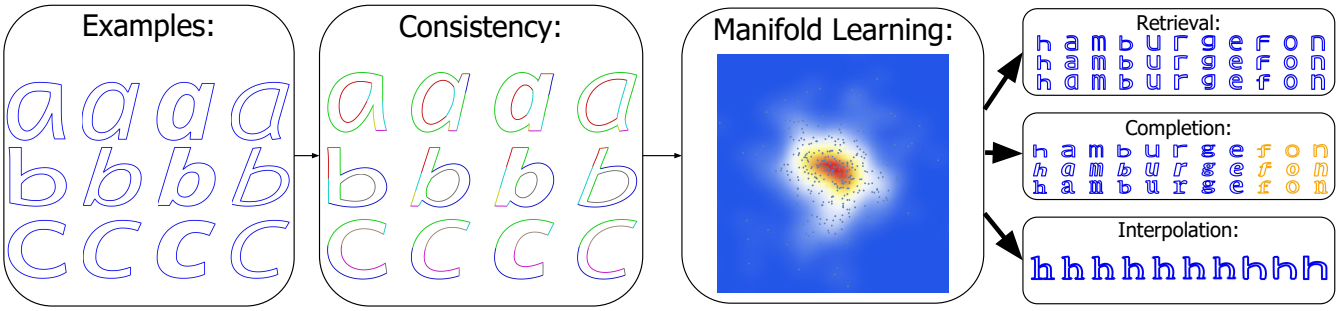


Figure 1: We learn style variations from existing typeface collections by representing them using a consistent parameterization, and projecting onto a low dimensional manifold. We start with a set of glyph examples that are not consistently parameterized, and use our fitting method to produce a part-based template parametrization consistent across same glyphs. We project the parametrization features into a low dimensional manifold and thus produce a missing-data-aware generative model. The resulting manifold can be used for exploratory applications to understand collections of fonts: topology-aware font retrieval, completion, and interpolation.

Abstract

Designing fonts and typefaces is a difficult process for both beginner and expert typographers. Existing workflows require the designer to create every glyph, while adhering to many loosely defined design suggestions to achieve an aesthetically appealing and coherent character set. This process can be significantly simplified by exploiting the similar structure character glyphs present across different fonts and the shared stylistic elements within the same font.

To capture these correlations we propose learning a stroke-based font representation from a collection of existing typefaces. To enable this, we develop a stroke-based geometric model for glyphs, a fitting procedure to re-parametrize arbitrary fonts to our representation. We demonstrate the effectiveness of our model through a manifold learning technique that estimates a low-dimensional font space. Our representation captures a wide range of everyday fonts with topological variations and naturally handles discrete and continuous variations, such as presence and absence of stylistic elements as well as slants and weights. We show that our learned representation can be used for iteratively improving fit quality, as well as exploratory style applications such as completing a font from a subset of observed glyphs, interpolating or adding and removing stylistic elements in existing fonts.

1. Introduction

Font collections are ubiquitous in design tools ranging from word processors to graphics editors. Although professional typographers perpetually create new and diverse families of fonts, it is not always possible to find a font with desired visual features, and thus designers often have to compromise by selecting a font that does not fit with their graphic message.

Designing glyphs that have desired style, aesthetic appeal, coherence, and follow good typographic practices requires substantial expertise and time investment. Even a small edit to one glyph (e.g., make the base of the 't' a little wider) can require subtle updates to many other glyphs to maintain coherence within the font. Since tools are not available to support these edits, font customization is generally inaccessible to all but a few expert font designers.

The goal of our work is to describe a part-aware font representation and show how it can be integrated with machine learning techniques to automate font manipulation. The described system is designed as a first step to assist non-experts in the design of customized fonts, it is not intended to compete with carefully designed high-quality fonts. We observe that priors on glyph structure and stylistic consistency are implicitly encoded in existing typefaces. We therefore hope to capture this knowledge in a generative model and use it to facilitate font editing, completion, interpolation, and retrieval.

The first challenge in learning from existing typefaces is that they do not share a common structure. Glyphs are typically represented by closed outline curves composed from primitives such as lines and Bezier curves with no consistency across different fonts even for the same character.

One approach to achieve a consistent representation would be to rasterize the glyph to a fixed-size image [USB16], however, this representation loses the advantages of vector graphics and does not allow preserving sharp features of the original glyph geometry.

Another option is to represent each glyph's outline with a polyline such that endpoints of each line segment are in correspondence across all glyphs [CK14]. Outline-based representation makes it difficult to reason about decorative elements such as serifs that are present only in a subset of glyphs. It also ignores the stroke-based structure of characters which is often consistent across glyphs and can be used to facilitate analysis. Phan et al. [PFC15] proposed a stroke-based glyph representation, however, this method still requires significant human interaction during the learning step to convert given outlines to the proposed representation, which hinders non-expert usage.

In this work we represent a glyph as a set of strokes and outline segments, where the latter are defined with respect to the coordinate system of their respective stroke. Many fonts are constructed with the appearance of brush strokes, to varying degrees; we believe they are better modeled by a spine and profile curves. A better model means more appealing interpolation and synthesis of characters as compared to raster or polyline representations. Consistency in stroke structure and low variance in profile offsets are among the key ingredients in our parametrization which make it easier to learn correlations between different parameters. This representation can also naturally handle discrete topological variations (i.e., addition and removal of strokes, such as serifs).

We take advantage of consistency in stroke structures of letters to create a common template for each letter. We propose an optimization procedure to align the template to the input outline, enabling re-parametrizing of any input glyph consistently with the other glyphs of the same letter. Unlike previous work on outline alignment [CK14], our method does not require joint analysis of the entire dataset, which makes our method scale linearly with respect to the number of fonts and allows us to process a collection of fonts an order of magnitude larger.

We formulate our representation as a concatenation of glyph descriptors for the entire font. Since there are co-dependencies in the concatenated glyph descriptors, due to stylistic similarities within a font and global structural similarities across fonts, it is possible to learn a reduced-dimensionality representation that captures the salient shape variations within a font collection. However, data is not always available for all glyphs (if only certain letters have been created by a designer for some font, or if the template fitting algorithm produces a poor fit). Therefore, we employ an EM-PCA algorithm [Row98] that is robust to missing data to learn the reduced-dimensionality representation. This model is a linear form of the more general Variational Auto-Encoder (VAE) [KW13] (we also experiment with nonlinear VAEs, and find that the linear EM-PCA model works better for our cases). The result is a projection to and from a latent font space that can be used for manipulation, interpolation, and generation of fonts.

We use a dataset of publicly available fonts to benchmark the quality of fitting our stroke-based representation to unstructured outlines and evaluate the reconstruction quality from the learned manifold. We demonstrate that we can consistently parametrize many existing fonts with our template, and use this parametrization to learn a common representation for fonts. We leverage our learned model for topology-aware font retrieval and for completing missing glyphs from a subset of outlines.

2. Related Work

Located at the intersection of graphics and information design, font design has been a well-studied field for decades [Zap87, Car95, Tsc98, Tra03, OST14, as8]. Our work is therefore informed by rich prior research on representation, analysis, and synthesis of fonts and shape collections. The most influential and relevant previous work is described below.

Font Representations Font glyphs are examples of complex 2D shapes that convey both meaning and appearance. There are generally three common ways to represent fonts: bitmaps, strokes, and outlines. Bitmap representations are easy to use and compatible with computer code, but do not scale without introducing artifacts. Stroke representations capture glyph essence well and are generally more compact [Gon98] (which is extremely advantageous to fonts with thousands of glyphs such as Chinese, Japanese, Korean (CJK) fonts [LZX16]), but are generally less expressive [JPF06]. Outline-based representations are more expressive [JPF06], but require hinting to counteract rendering artifacts [hin, HB91, Her94]. Still, outline fonts are a prevalent for representing Latin script typefaces in popular scalable computer formats [Mic16].

Typically, customizing a font requires tedious outline or stroke-editing, which is addressed by parametric font representations [Knu86]. In industry, a popular approach is to use master fonts, which enable high-level parametric control by interpolation with consistently placed control points [Ado97]. A recent OpenType format [Mic16] allows more elaborate parametrization during font design, for example, by continuously varying width or weight. The main limitation of these techniques is that they require typeface designers to provide a parametrization of the font. This might be possible within a small family of related fonts (i.e., a typeface), but these parametrizations are still inconsistent across typefaces. In contrast, we aim to achieve similar expressiveness while imposing minimal effort on typographers, simply by analyzing existing fonts.

To simplify the process of creating parametrized fonts, several systems have been proposed that prescribe a mapping between a few high-level font parameters and the control points of every glyph. For example, the Metafont system [Knu86] was used to create Computer Modern typeface families governed by a small set of parameters. These control continuous attributes, such as width and height of glyphs, as well as discrete variations, such as presence of serifs. Hersch and Bétrisey [HB91] described a set of rules to fit a topological model to outlines to enable automatic hinting. Shamir and Rappoport [SR98] proposed a visual tool for designing parametric fonts with constraints, and Hu and Hersch [HH01] provided

a richer set of geometric components to define a parametric typeface. Shamir and Rappoport [SR99] describe a procedure to represent outline-based oriental fonts using a hand-designed parametric font model and a procedure to compact the resulting representation using quantization. This work is different from our method in several ways: first, their parameterization requires manual assistance; second, their system fits deformable primitives to glyphs and does not aim for a consistent primitive set; it cannot be used for learning a manifold of fonts and for reasoning about part relationships within each glyph.

Learning Font Representation The first step in learning a font representation is establishing consistency across glyphs. One simple approach is to rasterize a glyph to a fixed-size image. This is effective for recognition [WYJ*15], but poses challenges for font synthesis. Current image synthesis methods yield blurry results, omit small features, and fail to enforce shape and curve continuity in the resolutions required for most fonts [USB16] (see Figure 11b).

Early work in this direction [Ada89] proposed decomposing glyphs into parts and deriving rules that share design attributes across glyphs to encourage consistency. In this system, all control glyphs have to be prescribed to create the complete font, and all relationships between prescribed and synthesized glyphs are defined manually. Campbell and Kautz [CK14] introduce a vector representation for learning font shapes, and learn a manifold of fonts. Their method is based on correspondences between glyph outlines. They jointly optimize for all control points by sliding them along the input outline while matching geometric features (e.g., normals and curvature) and preserving the original spacing (i.e., arc-length via elasticity objective). This approach does not explicitly recognize parts, and hence causes distortions in features like serifs when interpolating serif and sans serif fonts. In addition, this approach does not lend itself to learning the common shapes in different glyphs. For example the upper loop of 'g' and 'ɡ' are very similar, even though the overall outline is not. Similarly, decorative elements, such as serifs, need to be consistent in style, and together influence attributes of the glyph, such as its width. Please see [Ada86, Ada89] for extended discussion of common aesthetic principles. We allow glyphs with different topological structure to share a subset of strokes. This enables us to learn, for example, the shape of the upper loop in 'g' regardless of the shape of the bottom, and learn the shape of serifs only from glyphs that have the them. The resulting stroke-based template with potential topological variations can be matched to each glyph independently which enables us to avoid joint analysis and process an order of magnitude more fonts.

Phan et al. [PFC15] also propose to leverage stroke-based representation, modeling a glyph with brushes and caps aligned to canonical strokes. However, their method requires manual labeling of glyphs, limiting its applicability. It also represents each glyph as a set of stroke curves with profile offsets, a representation which suffers from artifacts in high curvature regions. Similarly, Lian et al. [LZX16] relies on a stroke/profile representation to generate hand-written fonts. Their proposed fitting procedure only focuses on stroke fitting and is very similar to our initialization [MSCP*07], but our contribution lies in optimization for Bezier control points which comes after stroke initialization. Finally, their system assumes that each type of stroke appears at least once in the input set and that the user provides at least several hundred characters, while our system does not have these requirements. Suveeranont and Igarashi [SI10] propose a weighted blend of outlines and skeletons to model new fonts as a linear combination of existing fonts. As they observe in their paper, linearly combining good fonts does not always yield a plausible result, since it does not capture co-dependencies between different parameters. Their solution is to limit the synthesis to a small set of the most similar examples, requiring a dense sampling of font space.

Analyzing 3D Shape Collections Our method is also related to existing work on analyzing 3D shape collections. In that domain, boundary-based consistent parametrization techniques [PSS01] were initially used to analyze similar surfaces with little topological variations. Part-based templates have been developed to handle more complex shapes [KCKK12, KLM*13], which enables a better analysis of topological variations [AXZ*15]. As with font analysis, they provide means to analyze relationships between objects that share only a subset of parts, and provide effective priors on global structure. Recently, several techniques have been proposed to analyze stylistic compatibility between objects [LKS15, LHLF15], aiming to separate content (i.e., object class) and style. Lun et al. [LKWS16] also propose a non-parametric method for transferring style by curve-based deformation, addition, and removal of parts. These approaches can be viewed as analogues to the problem of synthesizing fonts by transferring style from designer-created glyphs. The main advantage of our setting is that we can leverage a vast amount of data where the same content (i.e., letters) is presented in various coherent styles (i.e., each font is stylistically coherent). Of course, we provide a parametric model for glyph geometry, which is less challenging than arbitrary 3D shapes.

3. Overview

Typographers invest significant effort into designing glyphs that maintain a consistent style and adhere to good design principles. Thus, a well-designed font encompasses many implicit structural and stylistic relationships between the glyphs or their parts. The goal of this work is to learn these relationships from existing fonts and use them to simplify the design process of new ones. In particular, given a collection of fonts, we learn a low-dimensional representation that can be used to complete partial designs, interpolate between existing fonts, and provide easy controls to add and remove different stylistic elements, such as serifs.

To achieve this goal, a low-dimensional manifold of fonts needs to be learned across a variety of fonts. Since it is challenging to perform manifold learning on a general (inconsistent) font representation, we propose a two-step iterative process (Iterative Manifold Evolution, Section 7) to overcome these challenges: we parameterize fonts consistently by fitting the font model to all glyphs, and then we update the model using the newly parametrized fonts. This two-step process iterates until convergence to learn a font representation for all glyphs in a lower dimensional space.

We propose a stroke-based model that is concise, expressive, and is semantically driven — enabling us to factorize a glyph shape into meaningful elements. In particular, we represent each glyph with a set of strokes, where each stroke consists of a skeleton curve,

two outline curves on both sides of the skeleton, and a single cap curve for every endpoint. We represent all curves with Bezier control points. The skeleton control points of one skeleton segment are defined in the global coordinate system, while the other parts, the outlines and the cap are defined relative to that base curve or other skeleton segments in a tree-like manner (see Figure 2 right). This naturally separates profile features (e.g., width) from stroke features (e.g., slant), which facilitates learning relationships; outline control points will have a larger offset from the skeleton in a **bold** font and skeleton strokes will have a slope in an *italic* font. The stroke-based representation also naturally handles topological variations in glyphs. For instance, the glyphs 'g' and 'ɡ' can share parameters for the control points of the upper loop, but have different strokes to represent the tail. Similarly, decorative elements, such as serifs, can be represented with optional strokes. Thus, our consistent representation is defined with respect to a set of strokes, which we call a *deformable glyph template*, where template parameters include topological (which strokes are included) and geometric (control points of Bezier curves) ones (see Section 4).

To represent existing font outlines with our stroke-based model we propose a fitting procedure of the deformable template. Our fitting process takes advantage of the fact that stroke skeletons do not vary significantly across fonts, since letters are recognized by specific stroke structures. Thus we start by fitting the strokes of the template to the input glyph, followed by outline fitting, deforming the template Bezier curves to align with the boundaries of the input (see Section 5). Our template fitting process eliminates the need for joint analysis of all fonts, and is trivially parallelizable, which enables us to process 570 fonts, an order of magnitude more than previous techniques (such as Campbell and Kautz [CK14]). We focus our analysis on lowercase letters, as prior work highlighted that their design is more differentiated [Ada89], however, the system can be extended to uppercase letters and other characters by generating more templates.

To represent the entire font we concatenate individual glyph representations across all letters, generating a vector in a high-dimensional feature space. Since glyphs share a lot of stylistic and structural attributes, we expect this representation to be overcomplete, and learn a lower-dimensional font representation. For any given font, there will be missing entries: the feature vector including separate entries for different forms of 'g' and 'ɡ' (only one of which will appear in any given font), and serif parameters that do not occur for sans serif fonts. Thus, we will not have the entire feature vector during training or testing time. In addition, at test time, reasoning about the font representation by observing only a subset of the glyphs can give rise to many applications, such as font completion (see Section 9). We chose the Expectation Maximization Principal Component Analysis (EM PCA) approach [Row98], which provides a simple linear map for dimensionality reduction, and generalizes PCA to allow for missing feature values at test and training time (see Section 6).

4. Stroke-Based Deformable Template

Description We use a parametric deformable template $T_c(\Theta)$ to represent each letter c in the set of glyphs, where Θ are the deformation parameters that define the shape of the glyph generated

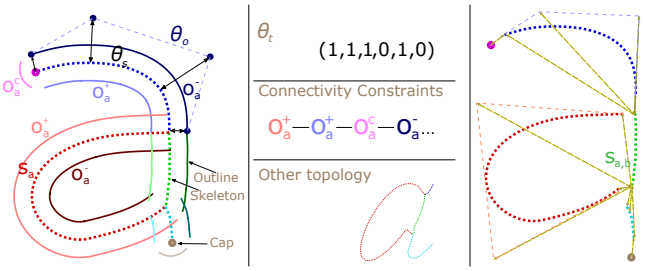


Figure 2: Deformable template representations. An example for the letter 'a'. Our model is described by parameters of strokes (θ_s), outlines (θ_o), and caps (left), as well as topological parameters (θ_t), indicating which strokes need to be present, and outline connectivity constraints (middle). Multiple topologies are prescribed for the same letter (middle, bottom). In each topology, one base skeleton segment ($s_{a,b}$) is represented in image space coordinates, while all other segments are offsets from it, or from other segments in a tree-like manner (right).

by $T_c(\Theta)$. Our template is described by a set of *skeleton segments* S_c and *outline segments* O_c, as shown in Figure 2 for example case when $c =$ 'a'. A single skeleton segment, named the *Base-Segment* $s_{c,b}$, is represented by the curve location in image space. The other skeleton segments are represented as offsets from $s_{c,b}$ or from another segment in a tree manner. Furthermore, every stroke corresponds to one skeleton segment $s \in S_c$, and each such segment corresponds to two outline segments on opposite sides of it: $\{o_s^+, o_s^-\} \in O_c$. We represent a *cap* as a special case skeleton stroke that has only one outline (o_s^c) associated with it. This dependency structure ensures consistent part and outline positioning during the fitting process, without the need to locally (or globally) align parts across different fonts. Note that one could also encode the template using angle relationships between segments, however we found positional differences to be more robust (as a slanted f still consists of horizontal wings), and produced better results for the learning step (see Section 6). Lastly, the template also prescribes the ordering of the outline segments on the generated outline and their connectivity constraints, as depicted in Figure 2 (middle). The connectivity constraints ensure the we always reconstruct a continuous outline, and practically remove redundancies from our representation. When a stroke is missing (e.g., a serif stroke in a sans-serif font), it creates a gap in the outline, which is trivially closed by connecting the two adjacent segments based on the ordering. In addition, Figure 2 (left) also demonstrates an example skeleton, its corresponding profile curve, and caps.

All curves in our framework are cubic Bezier, parametrized by their control points. Most skeletons and outlines are represented by a single curve, but long or more complex strokes are represented by two (which are enforced to be C^1 at the seam). The connectivity and smoothness constraints render some of the 4 Bezier control points redundant and thus they are removed from the representation. We denote the control points coordinates by $\theta_s \in \Theta_s$ for skeleton strokes and $\theta_o \in \Theta_o$ for outline curves. The control points of the outline curve are defined relative to the skeleton points, and skeleton points are defined relative to points on $s_{c,b}$ or other segments

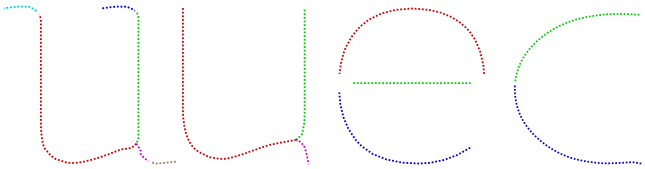


Figure 3: For each letter template, we provide a set of example layouts. Some depict topological variation (e.g. for the letter 'a', Figure 2). Some distinguish between decorative and sans-serif examples (e.g. 'u', left). Some templates include only one example (e.g. 'e', middle). These examples adhere to the connectivity and smoothness constraints defined by the template, for example the strokes of 'c' are constrained to be C^1 continuous (right). See the supplemental material for all template examples.

in a tree structure, as depicted by arrows in Figure 2 (left). This coordinate frame directly relates to meaningful properties such as brush width profile, and so is the natural choice for outline representation. Additionally, our template deformation parameters also allow discrete topological changes, which we denote by a vector of binary values, Θ_t. Each value corresponds to presence or absence of a single stroke. Thus, the shape of a deformed template is defined by the vector $\Theta = (\Theta_s, \Theta_o, \Theta_t)$. To facilitate fitting, we also provide a set of example topologies and geometries for each template, as demonstrated in Figure 2 (right bottom) and Figure 3.

Number of Templates For each character, we generally have two templates - for the serif and the sans-serif versions. Some characters (such as 'k') present significant variations in topology between fonts and hence warrant another template in order to be captured well, while others were successfully represented using one template only (such as 'o'). A general rule is to capture sufficient glyph variation to be used by the learning algorithm (see Section 8.1). The template generation process involves annotating junctions and part connectivity. We provide precise templates for every letter in the supplemental material.

5. Template Fitting

We now describe the template fitting procedure that finds the optimal deformation parameters for an input glyph $G_{c,f}$ of the letter c from a font f. Our fitting process is an optimization procedure in which we optimize for the parameters Θ such that the outlines of $G_{c,f}$ and $T_c(\Theta)$ become as similar as possible, in a consistent manner. The template T_c provides structure and geometric model for the glyph as well as an example for every possible topology of the glyph with approximate stroke shape: $E_c = \{\Theta_t^{c,i}, \Theta_s^{c,i} : i = 1..K_c\}$ (see Figure 3 for examples and all details in supplemental material). A demonstration of our process' consistency can be seen in Figure 4. In the following, we describe what are the considerations we account for when searching for a good fit (see Section 5.1), describe the iterative optimization procedure (see Section 5.2), and provide an initialization procedure for the template deformation process (see Section 5.3). Note that this fitting procedure is further refined through iteratively leveraging and evolving a manifold of the learned fonts, as described in Section 7.

5.1. Fitting Quality

In the following, we describe the objective function that evaluates the quality of a template fit. Our function favors shape correspondence between the input and the deformed template, alignment of feature points (sharp turns and corners), and avoids undesired local minima through regularization.

5.1.1. Correspondence

First, we penalize dissimilarities between the deformed template and the glyph:

$$E_{\text{corr}}(G_{c,f}, \Theta) = \sum_{x \in P(G_{c,f})} h_{\sigma_{\text{corr}}}\left(D(x, O_c(\Theta))\right) + \sum_{y \in P(O_c(\Theta))} h_{\sigma_{\text{corr}}}\left(D(y, G_{c,f})\right), \tag{1}$$

where $O_c(\Theta)$ is the outline generated by the current configuration (Θ) of the deformable template, $P(\Gamma)$ is a dense sampling of a curve Γ, and D measures point-to-curve distance. In practice, we sample 100 points per outline segment and use a KD-tree for nearest neighbor queries. We use a Gaussian kernel $h_\sigma(x) = 1 - e^{\left(\frac{x}{\sigma}\right)^2}$ of size σ_{corr} to reduce the influence of outliers and to better control the optimization behavior. The effects of this term is depicted in Figure 6 (bottom left).

5.1.2. Normal Consistency Regularization

Measuring the overall bi-directional distance between the two curves, in Euclidean space, can attract incompatible regions, sometimes as far as from the opposite side of the outline. Thus, we remedy this issue by favoring similar normal directions of the deformed template and the target outline it is projected onto. In particular, we augment our point representation with normal directions, lifting the 2D point $p = [p_x, p_y]$ on the target outline to 4D space: $[p_x, p_y, w_n n_x(p), w_n n_y(p)]$, where w_n is a constant weight, and $n(p)$ is the normal at point p. Specifically, for our deforming template outline, we lift each point q to become $[q_x, q_y, w_n n_x(q), w_n n_y(q)]$, and do the same for the target glyph $G_{c,f}$. With this augmented representation, we estimate the point-to-curve correspondence, and evaluate E_{corr} according to the $2D$ Euclidean distance of each point on $P(O_c(\Theta))$ from the one most relevant to it on the glyph in this combined position-and-normal space. Matching and regularizing normal directions is especially useful for avoiding local minima at outline corners, as it helps two sides of the outline to roughly preserve their original orientations while aligning to the input, instead of collapsing into one of the sides. This notion also further facilitates consistency, as it encourages the outlines to be similar to the skeletons, which are more similar to one another over different fonts.

5.1.3. Feature points alignment

Feature points, i.e. points of a large change in outline directions, are important for glyph appearance and provide a useful cue in the fitting process. Thus, aligning these points is essential in constructing a consistent database. We denote as $F(G_{c,f})$ the set of points which are the centers of such large direction changes. That is, if the curve changes its direction by an angle greater than $\frac{\pi}{6}$ over a short

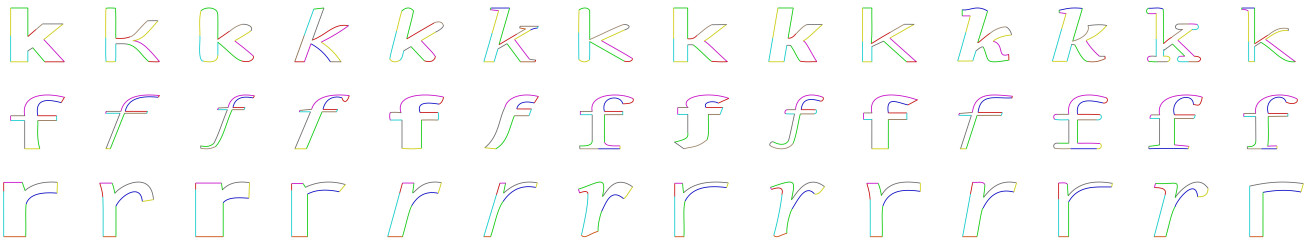


Figure 4: Consistency evaluation. Our template fitting process is focused on consistent semantic annotations of the input glyphs. The geometric and topological variety of three letters are depicted in each row, along with the optimization's successful capturing of the correct and consistent stroke arrangement for them.

distance, we mark the middle of this region as a feature point. Examining regions instead of point-wise changes in direction induces detecting both sharp and round corners.

To define these points on the template, we observe that sharp turns are more likely to occur at the boundary of two different strokes, thus we only consider junctions of outline segments as template feature points and denote them as $J(O_c(\Theta))$. This ensures that a specific outline segment does not undergo unnecessary turns in some glyphs compared to others, as demonstrated in Figure 6 (bottom right). To avoid foldovers in cases when feature points in G are close to one another in Euclidean space, but far-away along the curve, we measure the distances intrinsically rather than extrinsically for this term. Specifically, we define an intrinsic distance between a point $x \in J(O_c(\Theta))$ and feature point in $F(G_{c,f})$ as:

$$D_{\text{curve}}\left(x, F(G_{c,f})\right) = \min_{y \in F(G_{c,f})} \text{Arclength}\left(\text{Proj}(x, G_{c,f}), y\right), \quad (2)$$

where x is first projected to the glyph outline to be able to measure arclength:

$$\text{Proj}(x, G_{c,f}) = \arg\min_{y \in G_{c,f}} D(x, y). \quad (3)$$

Since we do not expect $J(O_c(\Theta))$ to be in perfect correspondence with the set $F(G_{c,f})$ we only match a subset of junction points that have a feature point nearby (i.e., $D_{\text{curve}} < \tau_{\text{ft}}$, $\tau_{\text{ft}} = 0.15 \cdot$ average segment length). Denoting this set as $F_{\tau_{\text{ft}}}(O_c(\Theta)) \subset J(O_c(\Theta))$, we can quantify feature alignment as:

$$E_{\text{ft}}(G_{c,f}, \Theta) = \sum_{x \in F(O_c(\Theta))} D_{\text{curve}}(x, F_{\tau_{\text{ft}}}(G_{c,f})), \quad (4)$$

5.2. Optimization

Starting with an initial template deformation Θ^0 we use gradient descent optimization to find the optimal deformation parameters. We found that directly optimizing the entire objective $E = E_{\text{corr}} + E_{\text{ft}}$ is prone to local minima issues and slow convergence. Thus, we opted for a procedure that alternates between optimizing E_{corr} and E_{ft}. We rely on correspondence matching in early gradient descent iterations and slowly increase the influence of feature-point compatibility term. This is controlled via r_{ft} parameter, the ratio of distance the feature points are allowed to be moved relative to their projected targets, which increases with every iteration. Similarly, we emphasize the weight of the normal matching with every iteration by modifying parameter w_n . In particular, the optimization parameters start with $\sigma_{\text{corr}} = 20, w_n = 25, r_{\text{ft}} = 0.3$ and end with $\sigma_{\text{corr}} = 5, w_n = 0.01, r_{\text{ft}} = 1$. We optimize for $N_{\text{iter}} = 70$ iterations and update the weights at every iteration by interpolating them linearly with respect to iteration count. Figure 6 demonstrates an initial guess and optimized fitting result with some terms of the objective function omitted.

5.3. Initializing Template Deformation

Starting with initial example strokes in E_c we now obtain initial guess for our fitting procedure, Θ^0. See Figure 3 for an example of a stroke structure in E_c. These approximate examples lack outline geometry and often are too dissimilar from input glyphs to serve as our initial guess for the outline optimization. Instead, we start by extracting a skeleton from an input glyph and fitting every example stroke structure to it. These stroke structures exhibit far less variance than the full outline, which naturally simplifies this step. Once we fit the stroke, it is simple to estimate a uniform width initial outlines around the strokes. We then pick the parameters that yield the best energy to define the initial state for the outline optimization, Θ^0.

5.3.1. Stroke Initialization, Θ_s^0

To fit the stroke parameters to an input glyph $G_{c,f}$ we first estimate its skeleton $S_{c,f}$. We use a method based on shape diameter function [SSCO08] to extract the major stroke skeletons defined by the outline. We then connect these regions by tracing a shortest path between them via medial axis points, forming a connected graph of skeleton paths, where most vertices have degree 2. This procedure provides a single connected curve network for the skeleton, which unlike the medial axis, captures only the main strokes, reducing sensitivity to small outline perturbations. See Figure 5a for an example output of this procedure.

Our next step is to fit the template strokes to the extracted skeleton. We found that directly deforming the template skeletons is prone to failure, so instead we segment and parametrize the extracted skeleton consistently with the template. Since strokes are continuous and smooth curves, and their junctions typically occur at sharp skeleton features, or at intersections with other strokes (or skeleton curves). We formulate the segmentation of the stroke according to the template through a CRF-based objective, where the nodes are skeleton points, the unary term is defined by registration

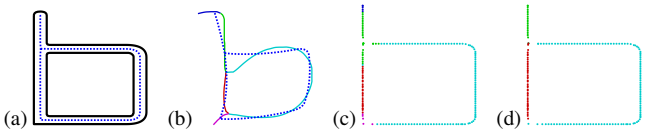


Figure 5: We illustrate our skeleton fitting pipeline. Given an input glyph, we first extract its skeleton (a). We then register this skeleton to a template (b) to obtain an initial segmentation (c). We then use CRF-based segmentation to obtain the final consistent segmentation of the skeleton with respect to the template (d).

of the skeleton to the template, and the pairwise term favors cuts at sharp features and intersections. In particular, we optimize:

$$E = \sum_{x \in V} W_u \cdot U(x, L(x)) + \sum_{x,y \in V, L(x) \neq L(y)} W_b \cdot B(x,y), \quad (5)$$

where the unary $U(x,l)$ term is a penalty for assigning a point x a label l and the binary term $B(x,y)$ is a penalty for a pair of adjacent skeleton points to have different labels. These are formulated as:

$$U(x,l) = h_{\sigma_u}(D_l(x)) \quad (6)$$

$$B(x,y) = \begin{cases} 1 - h_{\sigma_a}(A(x,y)), & (x,y) \in E \wedge x \notin J \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

To define the unary term, we use $D_l(x)$, a distance between the template and the extracted skeleton *after* it is registered onto the template skeleton using Coherent Point Drift method [MSCP*07], where the distance is measured to the l^{th} skeleton segment. To define the binary term we use the angle between the normals of adjacent points $A(x,y)$, where E denotes adjacency and J denotes junctions - points where more than 2 skeleton paths meet. Both terms are smoothed with the aforementioned Gaussian kernel h_{sigma}, and we set $\sigma_u = 0.25, \sigma_a = \sqrt{3}, W_u = 1, W_b = 4$.

After the skeleton is consistently segmented with the template, we parametrize every segment according to the template $T_c(\theta)$, i.e. we fit a Bezier curve to match each stroke $s_{c,f}$. Once the strokes of the glyph are fitted, we turn to initialize the outlines next.

5.3.2. Outline Initialization, Θ_0^0

We represent the outline in the coordinate system of the extracted skeleton segment (see Figure 2). To create an initial outline estimate, we first generate a profile curve at a constant distance that fits closely to the input outline for every stroke (i.e., such that the average distance between the curve and the closest point on the outline is minimized) (see Figure 6). The resulting outline is typically not continuous since the curve parameters are estimated independently for each stroke. We enforce C^0 continuity at junctions of outline segments by snapping the corresponding points to their average position. Sometimes this averaged control point can flip sides of the skeleton, which leads to inferior optimization performances. We detect these cases and project the average control point back to the correct side of the corresponding skeleton segment.

6. Learning

Given a collection of fonts $\mathcal{F} = \{F_1, F_2, ..., F_N\}$ we use our glyph fitting techniques to represent all letter glyphs consistently. We concatenate per-glyph parameters to create a feature vector representing a font: $F_i = [\Theta_{\cdot a'}, \Theta_{\cdot b'}, ...\Theta_{\cdot z'}]$. In this representation each font becomes a point in high-dimensional feature space: $F_i \in \mathbb{R}^{D_{\text{font}}}$, where $D_{\text{font}} = 3998$ (the number of parameters required to represent all glyph parts in a font, after enforcement of continuity and template part-sharing constraints). We expect this representation to be redundant due to stylistic and structural similarities in fonts: glyphs in the same font will share stylistic elements, and glyphs that correspond to the same letter will have similar stroke structure. We propose to learn these relationships by projecting all fonts to a lower-dimensional embedding. The main motivation behind this is to create a space where important correlations between different attributes are captured implicitly, and thus, any sample point X in this representation will implicitly adhere to common design principles and font structures in the training data $\mathcal{F}$. For this we need an embedding function M that projects a font to a low-dimensional feature space, i.e. $M(F) = X$, where $X \in \mathbb{R}^D$, and an inverse function C that can reconstruct a font from the embedding $C(X) = F$. To enforce learning correlations in the data we set the latent dimensionality $D = \alpha D_{\text{font}} = 39$ with $\alpha = 0.01$, which was determined experimentally to work best for our setup.

In addition to capturing important correlations, we want the embedding function to handle missing entries in the input feature vectors F_i because most fonts do not have glyphs with all possible topologies (i.e., decorative elements and stroke structures). In addition, in many applications in order to help the typeface designer we need to be able to reason about the whole font before it has been

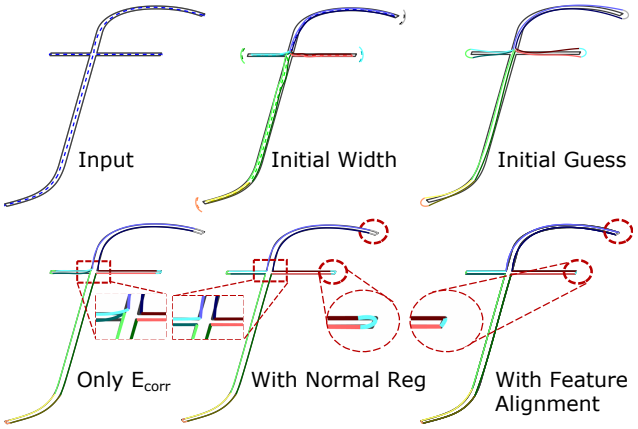


Figure 6: Outline optimization. Given an input glyph and its calculated skeleton (top left), we use its segmented version (see Section 5.3.1) to generate uniform width outlines around it (top middle). Our initial guess $O_c(\Theta^0)$ is the latter, after incorporating the template's connectivity constraints (top right). Our fitting procedure takes three considerations into account. Optimizing using only E_{corr} (see Section 5.1.1) can attract incompatible regions, as can be seen in the red box (bottom left). Considering normal directions (see Section 5.1.2) mitigates this issue (bottom middle). Finally, incorporating feature point alignment (see Section 5.1.3) is crucial for a semantically meaningful fitting, inducing consistent representation across different fonts. This is highlighted in the red circles (bottom right).

completed. To satisfy this requirement, we pick a linear embedding, with a slight abuse of notation: $M \in \mathbb{R}^{D_{\text{font}} \times D}$, and compute it from data points $\mathcal{F}$ via expectation-maximization principal component analysis (EM-PCA) method [Row98]. This generalization of PCA easily scales to large datasets and can handle missing data at train and test time. This method alternates between the **E-step** that optimizes for $\mathcal{X} = [X_1, ..., X_N]$, embedding coordinates of all fonts $\mathcal{F}$, and the **M-step** that optimizes for linear maps M, C. In this case C spans the space of first D principal components, and we can compute $M = (C^T C)^{-1} C^T$. To find an optimal linear map, the optimization starts with a random matrix M. In the **E-step**, we set $\mathcal{X} = M\mathcal{F}$, however, in case with missing data, each X_i and F_i are optimized for to minimize the norm: $\|CX_i - F_i\|$, where missing entries of F_i and entire vector X_i are free variables, this problem is then solved to find a least squares solution. In the **M-step**, we compute $C^{\text{new}} = \mathcal{F}\mathcal{X}^T(\mathcal{X}\mathcal{X}^T)^{-1}$. This process then iterates between the **E-step** and **M-step** K times to estimate the matrix M and all projections X (in all our experiments $K = 100$).

While some existing non-linear manifold learning algorithms also have variants that address missing data issues (e.g., GP-LVM [NFC07] and autoencoders [SKG*05]), we perform an empirical comparison and show that a simpler linear model works better with our dataset (see Section 8.1 for discussion).

7. Iterative Manifold Evolution

The initial manifold is learned by starting the fitting optimization using skeleton-based initialization (see Section 5.3.2), measuring the fitting error (see Section 8.1), and retaining only those fits with $err(\Theta) < T$ to estimate a manifold. Given this manifold, we project the font to the manifold and reconstruct it: $M^{-1}(M(F_i))$. We then use the resulting parameters to initialize the optimization procedure for every glyph with $err(\Theta) > T$ (i.e. completing the style of glyphs with high error). For example, in Figure 7, we show an example of a failed fit from a skeleton-based initial guess (template fitting commonly fails because the skeleton-based initialization for template parameters is too far from global minima), and the improved guess after manifold re-projection.

We iteratively estimate the manifold and re-project remaining high-error glyphs in each iteration. As can be seen in Figure 8, the overall fitting error decreases, and fraction of successful fits increases. All results presented in this paper correspond to the fifth iteration of this procedure.

8. Evaluation

8.1. Fitting Evaluation

We test our method on 570 fonts obtained from online database [Fon17]. Although more fonts are available, we restricted ourselves to normal monospace, sans serif, serif, and slab serif fonts, which still provides a data-set that is an order of magnitude larger and more diverse than what has been used in previous work. We fit our stroke-based representation to every glyph in this font collection, providing a consistent representation across all fonts. Figure 4 shows some example fits that result from the iterative fitting procedure.

To quantitatively evaluate the expressiveness of our deformable

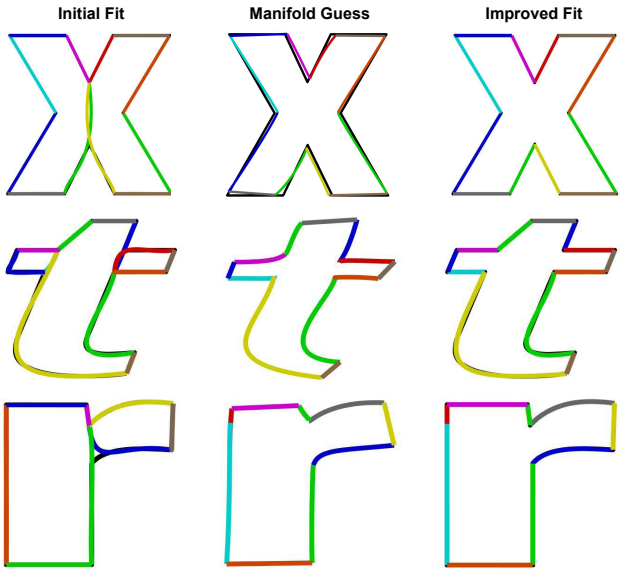


Figure 7: Sample fit improvements using iterative manifold evolution. Starting with a manifold-generated initial guess reduces the final fitting significantly in many cases.

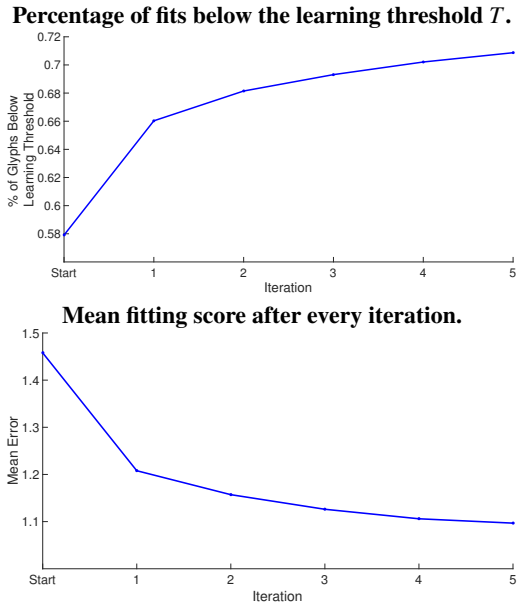


Figure 8: Iterative manifold evolution gradually increases the percentage of fits accepted for learning and reduces the mean fitting error.

template and accuracy of our fitting procedure we compare the outline generated by our template and the outline of the input glyph. To compare two curves we densely sample each curve so that the distance between consecutive points is 0.3 pixels (each glyph is scaled and represented to resolution of 200×200 pixels), and then for each sample we compute the distance to the closest point on

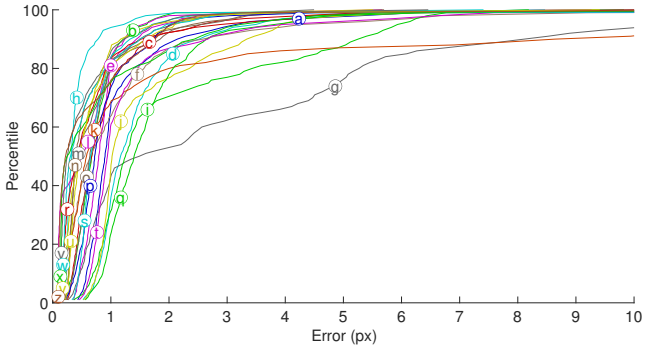


Figure 9: Fitting results by glyph. For all glyphs except for 'g', over 70-80% can be fitted well enough to be used for learning. 'g' is the hardest case to fit due to high variation in both topology and shape.

the other curve. We then average these distances, in a root-mean-square manner. We report these results in Figure 9, where the x-axis is an average fitting error threshold, and the y-axis is the fraction of glyphs that their average error is below the threshold. We observe that errors vary depending on glyphs, for example, 'g' poses the most challenges due to significance variance in its topology (e.g., the lower tail can be attached at various places on the upper loop and might have very diverse shapes). Figure 10 provides some representative visualizations of glyph appearance under different fitting errors.

To learn a manifold of fonts, we pick a conservative threshold of $T = 1.0$ pixels average error to ensure that we learn only meaningful correlations. At test time, however, we can still project any glyph to the manifold. Of course, the projection and reconstruction quality will suffer as the quality of fits degrades.

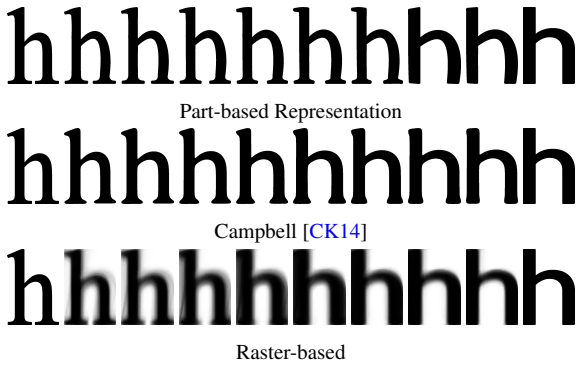
Figure 10: Glyph Fits at Different Degrees of Fitting Error. The threshold of $T = 1.0$ is picked as acceptable for learning purposes. During test time, worse fits might also be successful projected onto the manifold.

In the supplementary material, we provide an extended version of all our experiments over many of the fonts: We show the fitting result visualizations for a randomly selected sample of fonts, and the three application results ran over a larger set of fonts.

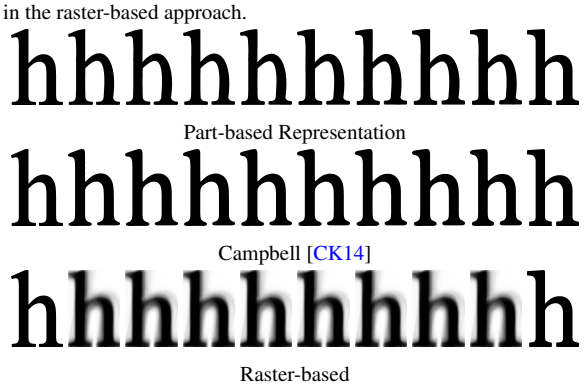
8.2. Comparison to Prior Work

We evaluate both the representation and the learning component of the method in comparison to previous work.

Representation Comparison We compare our method to the state-of-the-art method of Campbell and Kautz [CK14] and a raster-based approach. The goal of all three methods is to produce a consistent font representation, thus we evaluate each of the three representations using interpolation, which allows one to visualize



(a) Interpolations between fonts of different topology (TimesNewRomanPSMT and TrebuchetMS fonts). Interpolation between serifed and non-serifed fonts yields interpolations only between corresponding parts. Decorative elements, such as serifs, that are present only in one font are purposely not interpolated in our method, and the Bezier-based representation avoids the blurry artifacts seen in the raster-based approach.



(b) Interpolations between fonts of similar topology (TimesNewRomanPSMT and Georgia fonts). Quality of interpolation between same topology fonts is the same. Since our method enforces corner constraints during fitting, corners are better preserved during the interpolation process (seen on the sharp corners at the edges of the serifs).

Figure 11: Comparison of different representations.

qualitative changes in consistency as one transitions from one style to the other. Interpolation is the most direct way to measure consistency, while evaluation on other tasks, such as style prediction, will be confounded by properties of the learning algorithm.

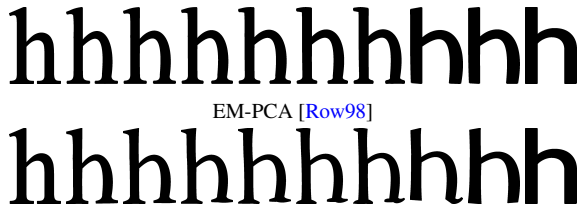
For both our approach and that of Campbell and Kautz [CK14], we employ the same interpolation scheme in a PCA space learned from the same set of 12 fonts. We investigated whether results for the latter approach differ when performed without PCA reduction, and found that the results are similar (see Supplementary Material). For the raster-based approach, we learn a variational auto-encoder [KW13] manifold on 64×64 rasterizations of glyphs with two fully connected layers of sizes 256 and 10, where the last layer performs sampling (as in [KW13]). Typically, neural networks require more training data, so to provide a fair comparison, we augment the training dataset with rasterizations from our full font dataset, since the datasets contained different fonts (to a total of 582 fonts). While results of the raster-based method will differ

with a different choice of training data or architecture, they still will exhibit blurry artifacts, as demonstrated in Figures 11a, 11b.

Unlike Campbell and Kautz [CK14], our model does not require that outline vertices on two glyphs of different styles but the same character have a bijective correspondence. This correspondence is ill-defined when comparing distinct topologies, as demonstrated in Figure 11a. This allows our method to explicitly interpolate between corresponding parts, which may be preferred to gradually degenerating serifs present only in one of the interpolants. In addition, explicit enforcement of corners from the fitting procedure often yields better preservation of sharp features in the resulting intermediate representations, as can be seen by the sharp corners of the ends of the serifs. On the other hand, the explicit correspondence aids in producing better quality results where this correspondence is indeed meaningful: for example, the intermediate interpolations of Campbell and Kautz [CK14] between two glyphs of the same topology better respect parallel constraints of the stems of 'h' (in Figure 11a) and generate slightly better width/curvature variation (in Figure 11b), compared to our method. Finally, both vector-based approaches outperform the raster-based method, which inherently produces blurry intermediate interpolations.

Generative Model Comparison To evaluate our generative model, we empirically compare it to a nonlinear variational autoencoder (VAE) [KW13] learned directly on the font representation from the fitting step. We implemented two variants: a vanilla version that learns on examples with consistent topology (no missing parts) and a denoising nonlinear VAE [VLBM08] that can handle different topologies and missing parts (please see supplementary material for training and architecture details). The vanilla nonlinear VAE produced smooth interpolations on examples in its domain (see Figure 12b). However, the denoising nonlinear VAE failed to produce meaningful interpolations between examples, introducing unnatural part deformations. This is likely due to the high-dimensionality of our model and the lack of training data samples (even after extensive data augmentation), which hinders the network from capturing the correct posterior distribution and learning a more complex model without overfitting. This phenomenon is often referred to as the curse of dimensionality [Bel15]. Additionally, learning on a cross-topology dataset requires imputing missing values (which are a form of noise), and, as has been highlighted [EHN*98, PPS03], linear methods may be preferable in low signal-to-noise ratio scenarios over non-linear models. Finally, the missing values in our case are "missing not at random" (MNAR), which is known to be challenging for existing methods [TLZJ17]. As a result, we found that for the considered scenario of MNAR and signal-to-noise ratio, a linear model is preferred. We thus use the a simpler linear model (EM-PCA) to demonstrate proposed applications, and leave analysis and development of more complex generative models (such as generalized nonlinear VAE [WHWW14] that additionally considers data relationships to mitigate dimensionality issue) for future work.

A parametrized font database and a low dimensional manifold enable a variety of exploratory applications that aim to browse and predict consistent glyph styles. Since it is challenging to quantitatively evaluate style, we follow the approach of prior work in showcasing a variety of qualitative experiments that demonstrate various applications of our method.



EM-PCA [Row98]

Denoising Nonlinear Variational Autoencoder (VAE)

(a) Interpolations between fonts of different topology (TimesNewRomanPSMT and TrebuchetMS fonts).

EM-PCA [Row98]

Denoising Nonlinear Variational Autoencoder (VAE)

Vanilla Nonlinear Variational Autoencoder (VAE)

(b) Interpolations between fonts of similar topology (TimesNewRomanPSMT and Georgia fonts).

Figure 12: Comparison of different generative models. Our method is able to produce meaningful interpolations between glyphs of same topology as well as glyphs of different topologies. The VAE method is only able to produce meaningful interpolations between glyphs of same topology (known as vanilla VAE). When applied to multi-topology data (denoising VAE), it introduces unnatural part variations.

9. Applications

Our font representation can be used in a variety of applications for modeling, organizing, and analyzing fonts. In this work, we focus on three applications that demonstrate the benefit of our method. First, our part-aware model enables the user to use topological constraints when retrieving the most similar font in a database (e.g., find a font that is most similar to this sans-serif font, but with serifs). Second, we demonstrate font completion results enabled by the use of an EM-PCA model that can handle missing data and the effective factorization of fonts into strokes and outlines in our representation.

To ensure that our results are representative of real-world use cases, we split the database into training (370) and testing (200) sets, where we learn our manifold on the training data, and use the test data as query either for retrieval or completion. For visualization, each glyph outline is drawn in its scaled unit box representation used for training (scale parameters may not be known during testing time).

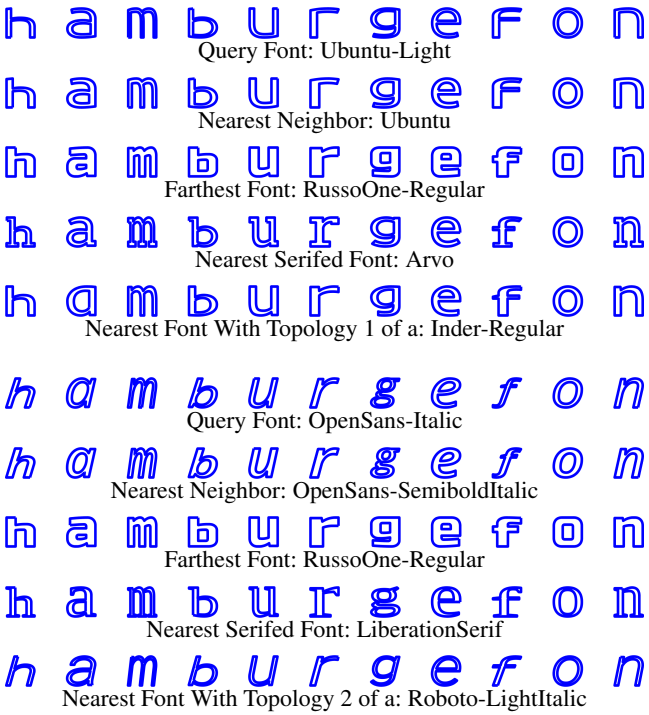


Figure 13: Font Retrieval. We project the query (testing) font into the manifold and search for existing fonts in the manifold. It can be seen that the nearest font is stylistically very similar, while the farthest font is quite distinct. We also demonstrate topology based queries, which take advantage of our part-based glyph representation. The style of the nearest fonts under these topology constraints also nicely matches the query font style.

Topology-aware Retrieval By consistently parametrizing a large variety of fonts and embedding them in a low dimensional manifold we implicitly organize font database and learn their similarities in context of the entire data-set. As was demonstrated before [CK14], this can facilitate font retrieval. In addition, our topology-aware representation enables us to constraint the retrieved results to have certain features (e.g., serifs, or particular stroke structure). In Figure 13 we present two queries of the database. For each query font (top row), we retrieve the most similar (second row) and dissimilar (third row) ones. We can also impose additional constraints on the retrieved font, such as it must have serifs (fourth row) or a different topology for some glyphs (fifth row).

Font Completion Since our learning method allows projecting a partial font vector to the manifold, we can reconstruct the full font from this a partial projection and thus complete the remaining glyphs. In this experiment, we hold out a subset of glyphs and use the manifold to predict it. In Figure 14 we visualize some sample completion predictions. Given a font consisting only of the glyphs in the word 'hamburgefon', we use the first seven letters to predict the style of the last three. It can be seen that our system correctly predicts the slant and part widths, italics, and decorative elements. Specifically, notice the consistency in slant in Figure 14(2,4,7,9).

Figure 14: Completion of fonts in different styles. The input glyphs are colored blue and the predicted glyphs are colored yellow.

The system is able to infer the correct variation of extremely thin styles such as (3) and (8), and thicker styles such as (2) and (5). Finally, notice that our system correctly predicted the existence of decorative elements in glyphs 'f' and 'n' examples (11-14), based on their presence in the the input glyphs.

An interesting completion question to ask is to see how many glyphs are needed to be designed in order to predict the rest. For example, one might want to design a subset of the most representative glyphs (such as the word 'handglove'), and complete the rest. In Figure 15, we complete the fonts UbuntuMono-Italic and AveriaSansLibre-BoldItalic, based on different numbers of given letters. As more and more letters are given, the prediction quality of some glyphs that generally exhibit a lot of variation improves (such as 'f' and 'm'), while the prediction quality of simpler glyphs such as 'x', 'y' and 'z' is correctly inferred even for only a small subset of provided glyphs. While the predicted glyphs are generally below the quality of manually designed fonts, the predictions suggest that the model does capture stylistic co-occurences necessary for generating a stylistically compatible set of glyphs.

Completion of UbuntuMono-Italic

Completion of Gudea-Italic

Figure 15: Completion robustness experiment. The number of letters necessary for completion is examined. The input glyphs are colored blue and the predicted glyphs are colored yellow.

10. Conclusion

This paper has proposed a new method for extracting a consistent parametrization for a large collection of fonts that can be used to assist font design applications. In comparison to previous work, the key novelty is the automatically extracted and part-aware font representation, that describes a glyph outline with respect to a segmented, center-line skeleton, which is suitable for learning a font manifold. We provide template fitting algorithms to extract this representation automatically and investigate manifold learning algorithms to factor co-occurences within and across fonts in this representation. We demonstrate that the low-dimensional manifold learned with the proposed methods is useful for a new font design application, font completion, as well as traditional applications like font retrieval and font interpolation.

This work is one step towards font design by non-experts. It provides the underlying framework for synthesizing new fonts within a manifold learned from data. However, it could be extended in many ways. First, it could be integrated with interactive tools that provide real-time feedback, user-provided constraints and postprocessing techniques that generate print-quality glyphs from coarse predictions of the learning step. Second, it could be extended to handle wider varieties of fonts and characters (for example, Cyrillic scripts). Finally, it could be adapted to handle data from other fields where 2D boundary curves must be co-parametrized and analyzed (e.g., medicine, agriculture, etc.).

11. Acknowledgements

We would like to thank Ariel Shamir, Princeton Graphics and Vision groups, and anonyous reviewers for suggestions and discussions and Asya Dvorkin for help with setting up experiments. During this work, Elena Balashova was supported by the National Science Foundation Graduate Fellowship (NSF-GRFP) and support was provided by NSF CRI 1729971 and Adobe.

References

[Ada86] ADAMS D. A.: *A dialogue of forms: letter and digital font design*. PhD thesis, Massachusetts Institute of Technology, 1986. 3

[Ada89] ADAMS D.: abcdefg(a better constraint driven environment for font generation). In *International Conference on Raster Imaging and Digital Typography* (1989), pp. 54–70. 3, 4

[Ado97] ADOBE: *Designing multiple master typefaces*. Tech. rep., 1997. 2

[as8] as8: Typography & type design. http://www.as8.it/edu/typography.html. Accessed: 2018-04-09. 2

[AXZ*15] ALHASHIM I., XU K., ZHUANG Y., CAO J., SIMARI P., ZHANG H.: Deformation-driven topology-varying 3D shape correspondence. *ACM Trans. Graph. 34*, 6 (Oct. 2015), 236:1–236:13. URL: http://doi.acm.org/10.1145/2816795.2818088, doi:10.1145/2816795.2818088. 3

[Bel15] BELLMAN R. E.: *Adaptive control processes: a guided tour*. Princeton university press, 2015. 10

[Car95] CARTER S.: *Twentieth century type designers*. Lund Humphries, 1995. 2

[CK14] CAMPBELL N. D., KAUTZ J.: Learning a manifold of fonts. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 91. 2, 3, 4, 9, 10, 11

[EHN*98] ENNIS M., HINTON G., NAYLOR D., REVOW M., TIBSHIRANI R.: A comparison of statistical learning methods on the gusto database. *Statistics in medicine 17*, 21 (1998), 2501–2508. 10

[Fon17] FONTS .: Fonts database. URL: http://www.1001fonts.com/. 8

[Gon98] GONCZAROWSKI J.: Producing the skeleton of a character. In *Electronic Publishing, Artistic Imaging, and Digital Typography*. Springer, 1998, pp. 66–76. 2

[HB91] HERSCH R. D., BÉTRISEY C.: Model-based matching and hinting of fonts. In *ACM SIGGRAPH Computer Graphics* (1991), vol. 25, ACM, pp. 71–80. 2

[Her94] HERSCH R. D.: Font rasterization: the state of the art. In *From object modelling to advanced visual communication*. Springer, 1994, pp. 274–296. 2

[HH01] HU C., HERSCH R. D.: Parameterizable fonts based on shape components. *IEEE Computer Graphics and Applications 21*, 3 (2001), 70–85. 2

[hin] Typotheque: Font hinting. https://www.typotheque.com/articles/hinting. Accessed: 2018-04-09. 2

[JPF06] JAKUBIAK E. J., PERRY R. N., FRISKEN S. F.: An improved representation for stroke-based fonts. In *Proceedings of ACM SIGGRAPH* (2006), vol. 4. 2

[KCKK12] KALOGERAKIS E., CHAUDHURI S., KOLLER D., KOLTUN V.: A probabilistic model for component-based shape synthesis. *ACM Transactions on Graphics (TOG) 31*, 4 (2012), 55. 3

[KLM*13] KIM V. G., LI W., MITRA N. J., CHAUDHURI S., DIVERDI S., FUNKHOUSER T.: Learning part-based templates from large collections of 3D shapes. *ACM Transactions on Graphics (TOG) 32*, 4 (2013), 70. 3

[Knu86] KNUTH D. E.: *METAFONT: the program*. Addison-Wesley Longman Publishing Co., Inc., 1986. 2

[KW13] KINGMA D. P., WELLING M.: Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013). 2, 9, 10

[LHLF15] LIU T., HERTZMANN A., LI W., FUNKHOUSER T.: Style compatibility for 3D furniture models. *ACM Transactions on Graphics (TOG) 34*, 4 (2015), 85. 3

[LKS15] LUN Z., KALOGERAKIS E., SHEFFER A.: Elements of style: Learning perceptual shape style similarity. *ACM Transactions on Graphics 34*, 4 (2015). 3

[LKWS16] LUN Z., KALOGERAKIS E., WANG R., SHEFFER A.: Functionality preserving shape style transfer. *ACM Transactions on Graphics 35*, 6 (2016). 3

[LZX16] LIAN Z., ZHAO B., XIAO J.: Automatic generation of large-scale handwriting fonts via style learning. In *SIGGRAPH ASIA 2016 Technical Briefs* (2016), ACM, p. 12. 2, 3

[Mic16] MICROSOFT: Opentype font variations. URL: https://www.microsoft.com/typography/otspec180/otvaroverview.htm. 2

[MSCP*07] MYRONENKO A., SONG X., CARREIRA-PERPINÁN M. A., ET AL.: Non-rigid point set registration: Coherent point drift. *Advances in Neural Information Processing Systems 19* (2007), 1009. 3, 7

[NFC07] NAVARATNAM R., FITZGIBBON A. W., CIPOLLA R.: The joint manifold model for semi-supervised multi-valued regression. In *2007 IEEE 11th International Conference on Computer Vision* (Oct 2007), pp. 1–8. doi:10.1109/ICCV.2007.4408976. 8

[OST14] OSTERER H., STAMM P., TYPOGRAPHIE S.: *Adrian Frutiger – Typefaces: The Complete Works*. Birkhäuser, 2014. 2

[PFC15] PHAN H. Q., FU H., CHAN A. B.: Flexyfont: Learning transferring rules for flexible typeface synthesis. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 245–256. 2, 3

[PPS03] PERLICH C., PROVOST F., SIMONOFF J. S.: Tree induction vs. logistic regression: A learning-curve analysis. *Journal of Machine Learning Research 4*, Jun (2003), 211–255. 10

[PSS01] PRAUN E., SWELDENS W., SCHRÖDER P.: Consistent mesh parameterizations. ACM, pp. 179–184. URL: http://doi.acm.org/10.1145/383259.383277, doi:10.1145/383259.383277. 3

[Row98] ROWEIS S. T.: EM algorithms for PCA and SPCA. In *Advances in Neural Information Processing Systems 10*, Jordan M. I., Kearns M. J., Solla S. A., (Eds.). MIT Press, 1998, pp. 626–632. URL: http://papers.nips.cc/paper/1398-em-algorithms-for-pca-and-spca.pdf. 2, 4, 8, 10

[SI10] SUVEERANONT R., IGARASHI T.: Example-based automatic font generation. In *Proceedings of Smart Graphics, LNCS* (2010), pp. 127–138. 3

[SKG*05] SCHOLZ M., KAPLAN F., GUY C. L., KOPKA J., SELBIG J.: Non-linear PCA: a missing data approach. *Bioinformatics 21*, 20 (2005), 3887–3895. 8

[SR98] SHAMIR A., RAPPOPORT A.: Feature-based design of fonts using constraints. In *Electronic Publishing, Artistic Imaging, and Digital Typography*. Springer, 1998, pp. 93–108. 2

[SR99] SHAMIR A., RAPPOPORT A.: Compacting oriental fonts by optimizing parametric elements. *The Visual Computer 15*, 6 (1999), 302–318. 3

[SSCO08] SHAPIRA L., SHAMIR A., COHEN-OR D.: Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer 24*, 4 (2008), 249. URL: http://dx.doi.org/10.1007/s00371-007-0197-5, doi:10.1007/s00371-007-0197-5. 6

[TLZJ17] TRAN L., LIU X., ZHOU J., JIN R.: Missing modalities imputation via cascaded residual autoencoder. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 1405–1414. 10

[Tra03] TRACY W.: *Letters of credit: a view of type design*. David R. Godine Publisher, 2003. 2

[Tsc98] TSCHICHOLD J.: *The new typography: a handbook for modern designers*, vol. 8. Univ of California Press, 1998. 2

[USB16] UPCHURCH P., SNAVELY N., BALA K.: From a to z: Supervised transfer of style and content using deep neural network generators. *arXiv preprint arXiv:1603.02003* (2016). 2, 3

[VLBM08] VINCENT P., LAROCHELLE H., BENGIO Y., MANZAGOL P.-A.: Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning* (2008), ACM, pp. 1096–1103. 10

[WHWW14] WANG W., HUANG Y., WANG Y., WANG L.: Generalized autoencoder: A neural network framework for dimensionality reduction. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops* (2014), pp. 490–497. 10

[WYJ*15] WANG Z., YANG J., JIN H., SHECHTMAN E., AGARWALA A., BRANDT J., HUANG T. S.: Deepfont: Identify your font from an image. In *Proceedings of the 23rd ACM international conference on Multimedia* (2015), ACM, pp. 451–459. 3

[Zap87] ZAPF H.: *Hermann Zapf & His Design Philosophy: Selected Articles and Lectures on Calligraphy and Contemporary Developments in Type Design, with Illustrations and Bibliographical Notes, and a Complete List of His Typefaces*. Society of typographic arts, 1987. 2