Supplementary Material

A. Training Detail

A.1. Network Architecture

Our SDF model is trained on 1 NVIDIA TITAN RTX GPU within 10G memory use and a batch size of 64 for 200 epochs in about 4 days. The learning rate of the SDF model is 0.0001, with an exponential scheduler for learning rate decay of 0.996 each epoch. The coefficient of the $\mathcal{L}_{\text{SDF}}$ is 3e3, $\mathcal{L}_{\text{Normal}}$ is 1e2, $\mathcal{L}_{\text{SDF}}$ is 5e1. The SDF model has HyperNetworks Ψ_S and ψ_S for glyph and font respectively, each consists of MLP with 4 hidden layers and 384 hidden dimension features, and with 192 dimension font and glyph latent code as input. The Φ_S has 5 hidden layers and 256 hidden dimension features. The activation function of Ψ_S, ψ_S and Φ_S are RELU. The network architecture of SDF Network is shown in Fig. B. The architecture of Hyper Networks is shown in Fig. A, and the same structure is applied in the later CF networks, which are uniformly denoted by Ψ or ψ in the subsequent figures.

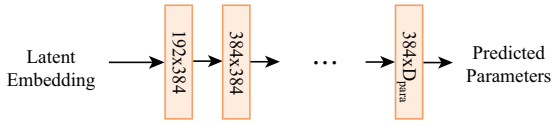

Figure A: Architecture of Hyper Network.

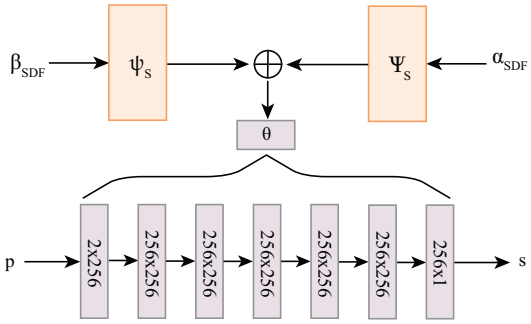

Figure B: Architecture of SDF Network.

Our CF model is trained on 1 NVIDIA TITAN RTX GPU within 5G memory use and a batch size of 64 for 500 epochs in about 2 days. The learning rate of the SDF model is 0.0001, with an exponential scheduler learning rate decay of 0.996 each epoch. The coefficient of the $\mathcal{L}_{\text{CF}}$ is 3e3, $\mathcal{L}_{\text{SDFlow}}$ is 3e3. The CF model has HyperNetworks Ψ_C and ψ_C for glyph and font respectively, each consists of 2 MLP with 1 hidden layer and 384 hidden dimension features, and with 192 dimension font and glyph latent code as input. The Φ_C has 6 layers and 256 dimension hidden features. The first HyperNet of Ψ_C and ψ_C are used to predict the parameters of the first three layers of the Φ_C, and the second is used to predict the parameters of the last three layers of the Φ_C. The activation function of Ψ_S and ψ_S are RELU, and the activation of Φ_S are Sine, which use the method mentioned in SIREN [1]. The network architecture of the CF Network is shown in Fig. C.

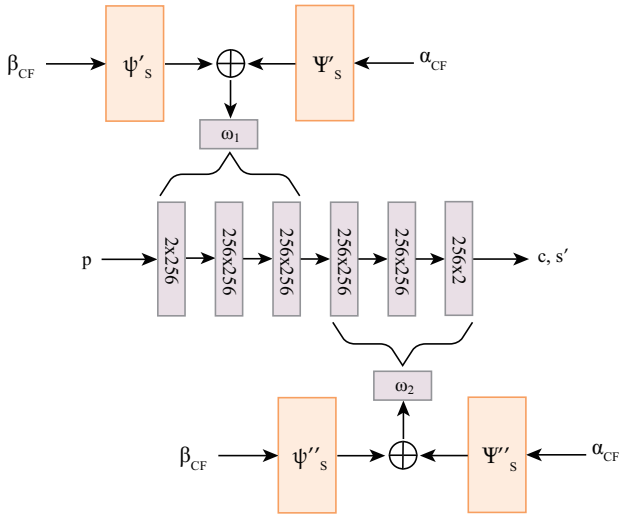

Figure C: Architecture of CF Network.

We also try different architectures, as shown in Fig. D. Experiments show that these two designs are worse than a network where each layer of parameters is derived from two latent codes (i.e., the network design of Fig. B and Figure Fig. C). When the number of network layers is particularly large, it is not reasonable to use Hypernets to predict the parameters of all layers, because this will cause the output dimension of the last layer of the MLP to be much larger than the input dimension, so this design of Fig. C should be followed in this situation.

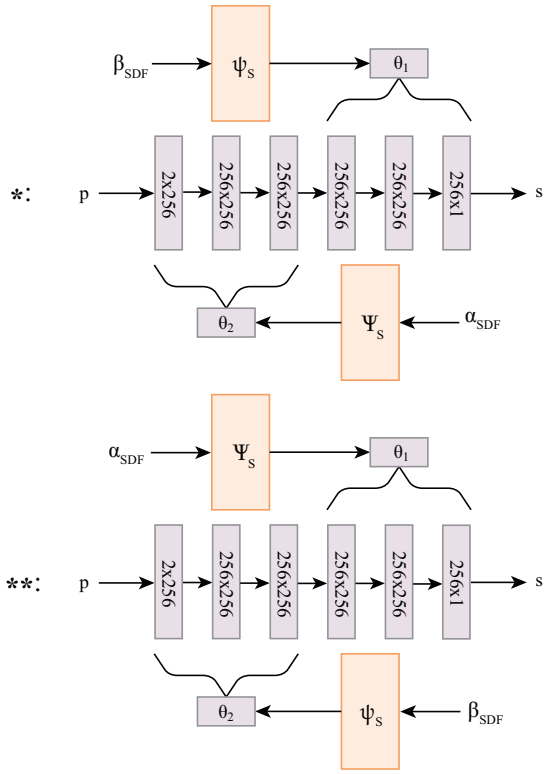

Figure D: Extra design of SDF Network.

A.2. Preparation for the training data

The dataset in DeepVecFont [2] has the vector sequence of different fonts (SVG paths) and a 64×64 low-resolution image of that sequence. We use only vector sequence (consisting of lines and cubic Bézier curves) data instead of low-resolution pixelated images when generating data.

We generated the following data for training from these vector sequences:

- For SDF Network: The coordinates of randomly sampled points on the $[0,1]^2$ and their SDF values $\bar{s}$; The coordinates sampled on the ground truth vector curves (their SDF values are 0), and their corresponding normal $\bar{n}$.
- For CF Network: The corners coordinate in each glyph; The coordinates of randomly sampled points on the $[0,1]^2$ and their CF values $\bar{c}$; The coordinates sampled around the corners and their CF values; The coordinates of randomly sampled points on the $[0,1]^2$ and their SDF values for SDFlow supervision $\bar{s}'$.

We sampled 5000 points on the $[0,1]^2$ and on the vector curves respectively for training. We first render the pixelated image of the entire font from the vector curves in high resolution of 1024^2 to ensure an accurate SDF supervision, then use the chamfer distance of the pixel map to calculate the SDF value, which gives the SDF value of each pixel on the image and then use bilinear interpolation in sampling to derive the SDF value of any coordinate point. When sampling on the vector curve, for each SVG path we do a numerical integration to calculate the path length and assign a different number of samples to each path weighted by these lengths. But when sampling we sample the parametric curve linearly from 0 to 1 for t in Bézier curves or lines. Although this is not uniform for the cubic Bézier curves, it does not introduce much error, and because we have enough samples, the sampling points densely cover the entire vector font profile.

For each parametric curve, we obtain the direction of its tangent by taking the derivative of its x and y components separately and taking the unit vector in the direction perpendicular to the direction of the tangent as the direction of the normal vector. And since there are two opposite directions of the vertical vector in the tangent direction, only one of them is the normal vector, so we check the pixel value after a very small distance (say 1 pixel) from the tangent point along the unit normal vector we obtained, and if it is black (inside the font), then we invert the obtained normal vector, otherwise, we keep the obtained normal.

For CF, we sampled 5000 points on the $[0,1]^2$ and their corresponding CF value for training. And for each glyph, we generate its corners by the smoothness at the connection of each path. We could calculate the tangents $\mathbf{t}_1, \mathbf{t}_2$ on the connection with respect to two paths on both sides of it. If the cosine similarity of this tangent is larger than a threshold:

$$1 - \langle \mathbf{t}_1, \mathbf{t}_2 \rangle > 0.01 \tag{1}$$

We accept the connection as a corner. For each corner, we sample 100 points in a box centered on it with a width of 0.1. SDFlow's data is generated in the same way as SDF's data.

Note that some of the first and second control points of the cubic Bézier curves in the data may overlap, and even more control points overlap, leading to numerical problems, so it's better not to derive at t = 0 or 1 to obtain the tangent on the start or the end of the control point, but $t = \delta$ or $1-\delta$, where $\delta = 0.001$.

B. Vectorization Detail

B.1. Extracting Ordered Point Set

The primary task of vectorization is to extract the ordered point set on the contour from the SDF. As previously mentioned in the paper, we first render a binary pixel image and perform an edge detection: the contour points would be the black pixel whose Manhattan distance to the nearest white pixel is less than 2 (there is a white pixel in its 8 pixels surrounding), and we denote these pixels as **Contour Points**.

Until this step, we only get an unordered set of points, next we'll sort them using search.

There are many ways to traverse these Contour Points, the desired order is to start at a point (commonly has the maximum x or y value in this connected block) and traverse all contour points in the same connected block as it, and then traverse the last point adjacent to it (i.e., traverse one turn back to itself). Since the path we fit is a closed path, and a closed path necessarily has loops, each connected block is necessarily composed of one or more loops, and in general, only one loop.

In the case of a connected block with only one loop, we only need to use depth-first traversal from one point to traverse the entire loop. In the case of multiple loops, direct depth-first or breadth-first traversal can be problematic. So in this case, we split the loops and then traverse each split single loop.

B.2. Obtain the Normal of Contour

Before performing the dual contouring, we have to estimate their tangent vectors and normal vectors for the obtained discrete contour points. We calculate the tangent direction on the target contour point by averaging multiple lines of contour points with equal order difference between the left and right of the target contour point (e.g., the 5th point on the left and the 5th point on the right of the target point are connected, and the 6th point on the left and the 6th point on the right of the target point are connected, and these two lines are averaged as the tangent direction). Of course, other methods must exist to estimate the tangent direction. The advantage of this method is that the tangent or normal vector is smoothed and is not affected by the high-frequency fluctuations of the surface brought about by the neural network fitting, which in turn also allows estimating the smoothed surface curvature and doing dual contouring from the point of maximum surface curvature will generally give better results.

B.3. Vectorization Process

This section is going to talk about the vectorization on $\{\mathbf{c}_s, \mathbf{c}_e, \{\mathbf{p}_i\}\}$, where $\mathbf{c}_s$ is the corner start, $\mathbf{c}_e$ is the corner end, and $\mathbf{p}_i$ is a set of contour points between $\mathbf{c}_s$ and $\mathbf{c}_e$, and the results is a set of splines $\{s_1, s_2, ..., s_n\}$. From the first point of $\{\mathbf{p}_i\}$, we maintain a set $\mathbb{S} = \varnothing$ initially, constantly add a point into $\mathbb{S}$, and use a Line or a Quadratic Bézier curve to fit points in $\mathbb{S}$ with minimum fitting error.

Fitting the Lines We start fitting s_1 by assuming the temporary spline to be fitted is a line. Initially, we take the $\mathbf{p}_0$ as the start control point of the splines we want to fit. Progressively, each time we add the temporary $\mathbf{p}_i$ in, we consider it to be the end control point of the straight line $\mathbf{L}$ we want to fit, and we calculate the maximum error E_{L} of the distance from the point in $\mathbb{S}$ to the line. If it exceeds the threshold, then there are two types of processing:

1. If $\|\mathbf{L}\|_2 \geq l_{\min}$, output the $\mathbf{L}$, empty the $\mathbb{S}$, and fit next spline.
2. If $\|\mathbf{L}\|_2 < l_{\min}$, we consider the points in $\mathbb{S}$ are on a Quadratic Beziér curve $\mathbf{Q}$, and start fitting this curve.

Moreover, if the temporary spline s_i to be fit is not s_1, then we should check the tangent on the last control point of the previous spline s_{i-1}, and check the slope of the temporary line to be fitted, if their angle is greater than a certain threshold $\mathrm{th}_{\mathrm{smooth}}$, then move to the quadratic curve fitting step.

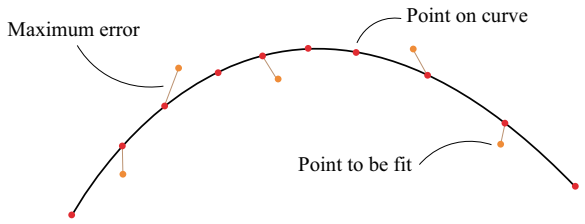

Figure E: Fitting error between a set of coordinates and a quadratic Bézier curve.

Fitting the Quadratic Curves Before fitting the quadratic curve, we first calculate the maximum distance of the points in the set to the quadratic curve to estimate the error. Suppose we want to calculate the shortest distance from the point in $\mathbb{S}$ to the quadratic curve $\mathbf{Q}$. And for the quadratic Bézier curve, although it is possible to obtain an analytical solution in the form of solving a cubic equation for the shortest distance from a point to a curve, that would be a bit complicated. Instead, we obtain a set of a bunch of points $\mathbb{Q}$ on $\mathbf{Q}$ by Taking $2|\mathbb{S}|$ points at equal intervals with respect to t on $\mathbf{Q}$, and then compute the maximum distance from the points in $\mathbb{S}$ to the points in $\mathbb{Q}$ in O(t) complexity to make an approximate estimate of the error, as shown in Fig. E.

For the Quadratic Bézier curve, since we have determined the start and end control points, the only thing left to do is to determine the second control point. In addition, we know that there are no corners between $\mathbf{c}_s$ and $\mathbf{c}_e$, so the splines we fit must be smoothly connected. There is a good property of the Bézier curve that its tangent on the 1st control point is the direction of its 1st control point to the 2nd control point. Thereby, we just need to find the second control point along the tangent direction of the control point at the end of the previous curve, we achieve this gold by a dichotomous search.

We use the start control point as the left endpoint $\mathbf{d}_l$ of the dichotomous search, and the right endpoint $\mathbf{d}_r$ of the

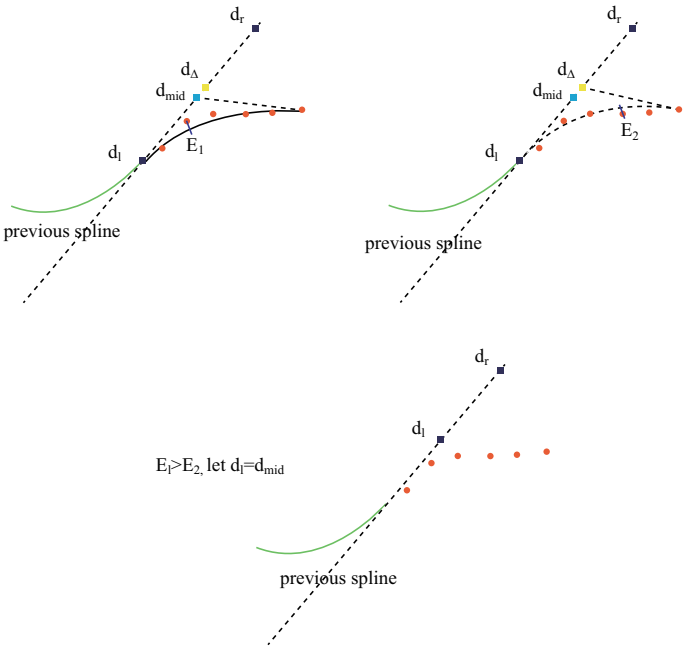

Figure F: One of the cases of dichotomous search.

search is a distance from the start control points along the direction of tangent $\mathbf{t}_{i-1}$ of the end control point of the previous spline. We find the midpoint $\mathbf{d}_{\text{mid}}$ of $\mathbf{d}_l$ and $\mathbf{d}_r$ for each search, and another point $\mathbf{d}_\Delta$ along a small distance from $\mathbf{d}_{\text{mid}}$ to $\mathbf{d}_r$. Then we calculate the fitting error E_1 between the $\mathbb{S}$ and $\mathbf{Q}_1$, where $\mathbf{Q}_1$ is the quadratic Bézier curve takes $\mathbf{d}_{\text{mid}}$ as the second control point. And we also calculate the fitting error E_1 between the $\mathbb{S}$ and $\mathbf{Q}_2$, where $\mathbf{Q}_2$ is the quadratic Bézier curve takes $\mathbf{d}_\Delta$ as the second control point. And if $E_1 > E_2$, we let $\mathbf{d}_r = \mathbf{d}_{\text{mid}}$, otherwise we let $\mathbf{d}_l = \mathbf{d}_{\text{mid}}$. We will repeat this process until $\mathbf{d}_r - \mathbf{d}_l < \text{th}_d$, where th_d is the threshold indicating the dichotomous search will terminate below this search interval. Fig. F demonstrate an example of one of the cases of this process. After that, we output $\mathbf{d}_{\text{mid}}$ as the second control point of the current spline with minimum fitting error. Similarly, when this minimum error is greater than a certain threshold, we output the curve $\mathbf{Q}$ and clear $\mathbb{S}$.

Insert the $\mathbf{c}_s$ and $\mathbf{c}_e$ After obtaining series splines from s_1 to s_n, we need to insert the corner point that the first control point of the first spline is $\mathbf{c}_s$ and the last control point of the last spline is $\mathbf{c}_e$. Start from $i = 1$, constantly check the type of temporary spline s_i:

1. If s_i is a line, then make the $\mathbf{c}_s$ the first control point of it. If $i = n$ or the angle between s_i's tangent and the tangent of s_{i+1} is lower than $\text{th}_{\text{smooth}}$, then confirm the i for $\mathbf{c}_e$'s insertion. Otherwise, $i = i + 1$, and continue checking s_{i+1}.
2. If s_i Quadratic curve, then make the $\mathbf{c}_s$ the first control point of it. Confirm the i for $\mathbf{c}_e$'s insertion.

For $\mathbf{c}_e$, the process is similar. We start from $j = n$, constantly check the type of temporary spline s_j:

1. If s_i is a line, then make the $\mathbf{c}_e$ the last control point of it. If $j = i$ or the angle between s_j's tangent and the tangent of s_{j-1} is lower than $\text{th}_{\text{smooth}}$, then confirm the j. Otherwise, $j = j - 1$, and continue checking s_{j-1}.
2. If s_i Quadratic curve, then make the $\mathbf{c}_e$ the last control point of it. Confirm the j.

Finally, output the $\{s_i, ..., s_j\}$ as the acquired spline(s).

C. Additional Experimental Results and Discussion

C.1. Reconstruction

Fig. J and Fig. J show some reconstruction results in addition to the result in the main text. Each of the two lines in the diagram represents 52 upper and lower case letters of one of the fonts.

C.2. Interpolation

Fig. L and Fig. M show some interpolation on more complex fonts in addition to the result in the main text, where the first and last lines of every 5 lines represent a font in the dataset, Fig. L shows the case of these fonts in upper case and Fig. M shows the corresponding case in lower case. As can be seen from Fig. M, although there are variants between letters (e.g., lowercase 'a' and 'g'), or even some fonts with only uppercase, we can still achieve a reasonable interpolation between these glyphs.

C.3. Additional Explanation on Ablation Study

Tab. 1 shows the L1 error with and without the $\mathcal{L}_{\text{Peak}}$, Tab. 2 shows the divergence (chamfer distance between interpolated corner and original corner in the dataset) with and without the $\mathcal{L}_{\text{SDFlow}}$. We put the average number of corners for each letter in the dataset in the table (first row of each group) as a comparison. As can be seen in Tab. 1 the results with $\mathcal{L}_{\text{Peak}}$ are consistently better than those without it, and the improvement consistently increases as the number of corners in the dataset increases.

However, in Tab. 2, part of the letter with $\mathcal{L}_{\text{SDFlow}}$ has lower divergence than without it. This might happen due to the randomness of the interpolation rate, as shown in Fig. G, (d) to (h) demonstrate the interpolation of CF corresponding to the original glyph (a) and (c). (b) mark out the peak of CF of these CF, each color corresponds to one of the CFs, from (d) to (h): black, blue, green, red, cyan, with linear interpolation factor: 0, 0.25, 0.5, 0.75, 1. It can be seen from the

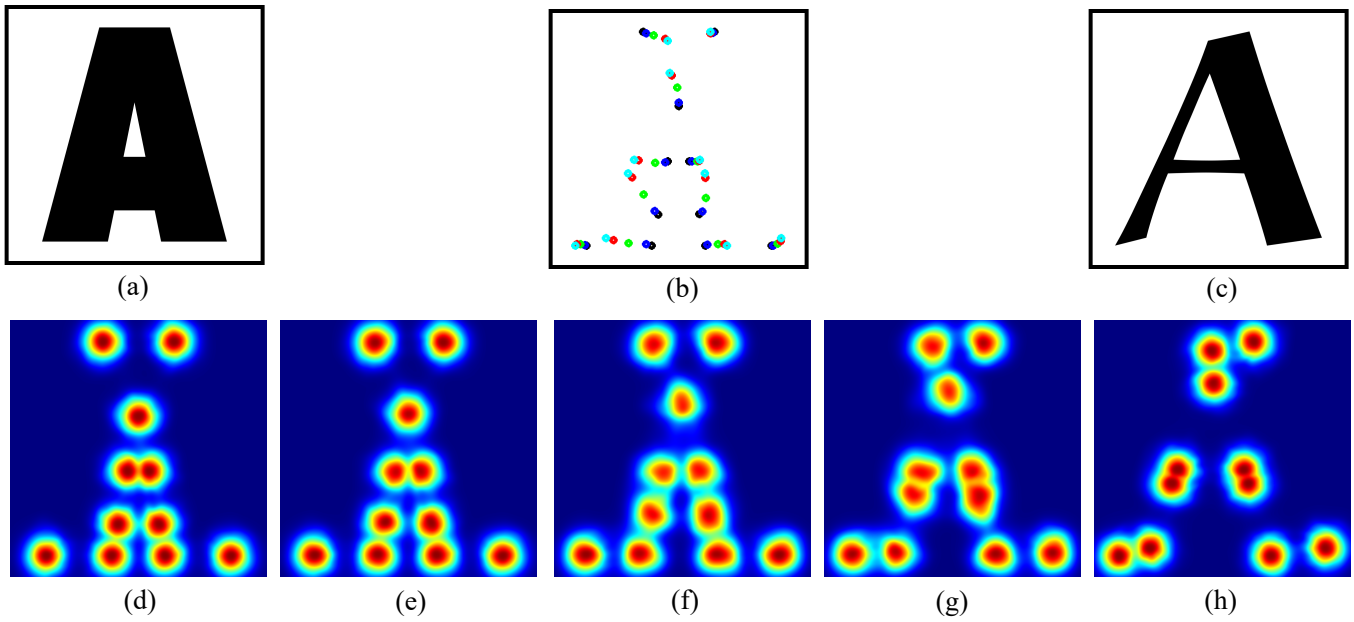

Figure G

figure that the distance between these different color points is not linear, this interpolation property is determined by the network, and linearizing the interpolation or controlling the interpolation rate will be an important topic in future work. But overall, SDFlow improves the interpolation quality in general, which could be more easily seen in Fig. 4 in the main text than in the divergence metric.

C.4. Style Inference and Font Shape Completion

Also as mentioned in the main text (Fig. 6), optimizing a latent coding in a CF network brings ambiguity, however, optimizing a latent code with an SDF network is viable, thus we provide the results on font generation tasks in Fig. H and Fig. I. Our models are able to do these tasks directly or indirectly, while we did not focus on them in the main submission because our focus is to sharpen corners.

For the few-shot style generation task (Fig. I), we freeze the parameter of the network and compute the ground-truth SDF value of the target style raster image $\mathcal{I}$ as the input of the network. We first initialize a latent code filled with 0, as the input style code α_{SDF}, fetch the glyph code β_{SDF} corresponding to the given image $\mathcal{I}$ from the trained codebook from SDF Net, and optimize the α_{SDF} for the inferred style. After obtaining the α_{SDF}, we combine it with β_{SDF} of other glyphs to synthesize the full font set.

For the shape completion (Fig. H), we generate the SDF value of the target incomplete raster image $\mathcal{I}$, and use the same process as the few-shot generation to fit a style code

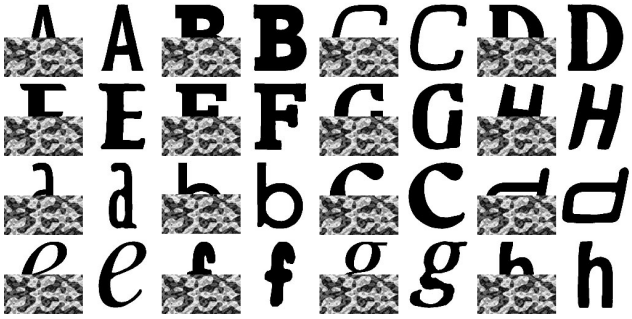

Figure H: Several different glyph shape completion on different font styles. The bottom half of the ground-truth fonts to be completed are masked.

α_{SDF}. The finally inversed code will be able to generate the complete shape.

References

[1] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[2] Yizhi Wang and Zhouhui Lian. Deepvecfont: synthesizing high-quality vector fonts via dual-modality learning. *ACM Transactions on Graphics (TOG)*, 40(6):1–15, 2021.

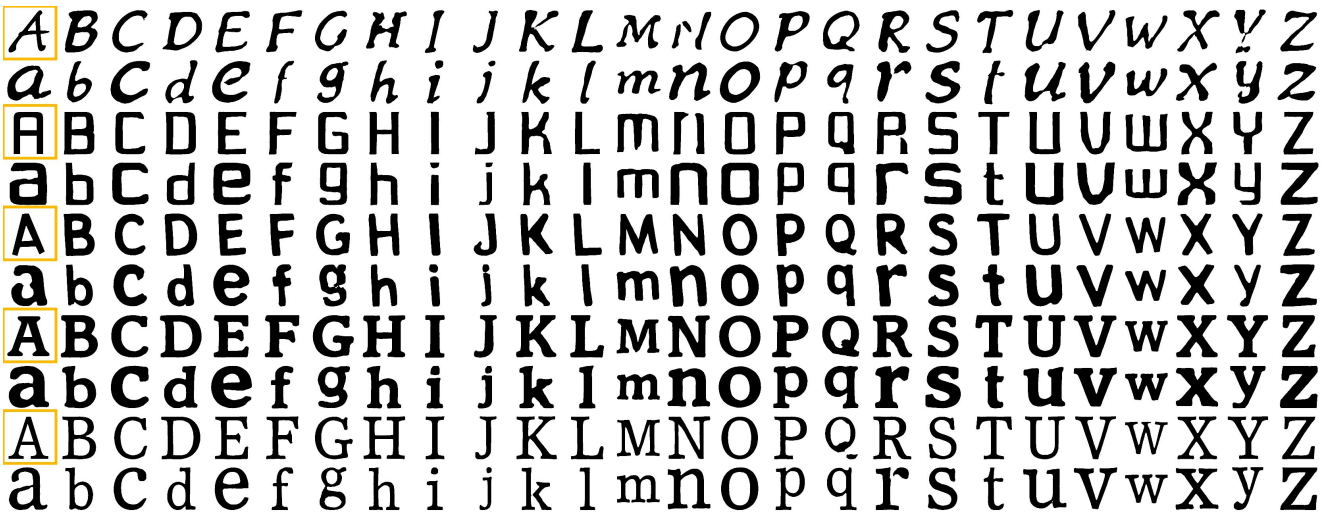

Figure I: One shot generation. The first 'A' in every 2 row represents the given GT font style.

Glyph	A	B	C	D	E	F	G	H	I	J	K	L	M
Corners	12.25	9.77	5.95	6.24	12.90	11.53	9.07	14.17	6.08	5.78	13.85	7.13	14.60
w/o $\mathcal{L}_{\text{Peak}}$	0.86	0.73	0.44	0.37	0.81	0.73	0.67	0.91	0.35	0.40	1.13	0.38	1.54
w $\mathcal{L}_{\text{Peak}}$	0.72	0.64	0.36	0.31	0.69	0.60	0.55	0.73	0.29	0.33	0.95	0.33	1.35
Glyph	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Corners	11.41	2.79	8.13	7.75	11.56	7.13	9.11	6.85	8.54	14.67	14.67	10.92	10.14
w/o $\mathcal{L}_{\text{Peak}}$	0.99	0.20	0.50	0.65	0.82	0.56	0.56	0.46	0.63	1.68	1.06	0.71	0.65
w $\mathcal{L}_{\text{Peak}}$	0.89	0.18	0.43	0.56	0.70	0.48	0.47	0.38	0.56	1.46	0.90	0.61	0.56
Glyph	a	b	c	d	e	f	g	h	i	j	k	l	m
Corners	8.83	8.28	5.47	8.04	8.04	11.55	8.67	10.17	6.43	6.42	13.12	5.46	13.08
w/o $\mathcal{L}_{\text{Peak}}$	0.59	0.69	0.31	0.65	0.48	0.74	0.75	0.77	0.37	0.50	1.18	0.34	1.15
w $\mathcal{L}_{\text{Peak}}$	0.49	0.60	0.25	0.55	0.40	0.60	0.63	0.67	0.30	0.40	0.99	0.28	0.95
Glyph	n	o	p	q	r	s	t	u	v	w	x	y	z
Corners	9.27	2.73	8.70	8.47	8.91	6.76	10.60	8.15	8.11	13.74	13.93	9.78	9.88
w/o $\mathcal{L}_{\text{Peak}}$	0.64	0.15	0.67	0.67	0.60	0.43	0.60	0.52	0.51	1.32	0.86	0.73	0.54
w $\mathcal{L}_{\text{Peak}}$	0.55	0.13	0.59	0.57	0.53	0.35	0.47	0.45	0.45	1.15	0.73	0.64	0.46

Table 1: Average number of corners for each alphabet in dataset, L1 error w and w/o peak loss.

Glyph	A	B	C	D	E	F	G	H	I	J	K	L	M
Corners	12.25	9.77	5.95	6.24	12.90	11.53	9.07	14.17	6.08	5.78	13.85	7.13	14.60
w/o $\mathcal{L}_{\text{SDFlow}}$	18.44	18.98	19.99	20.40	19.75	19.08	22.09	18.06	13.18	18.01	18.92	20.15	20.99
w $\mathcal{L}_{\text{SDFlow}}$	19.17	20.11	20.70	22.91	21.09	20.84	21.31	19.64	13.37	17.60	19.42	21.43	21.33
Glyph	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Corners	11.41	2.79	8.13	7.75	11.56	7.13	9.11	6.85	8.54	14.67	14.67	10.92	10.14
w/o $\mathcal{L}_{\text{SDFlow}}$	20.02	0.49	18.72	22.89	19.99	18.83	19.22	17.79	19.63	20.62	16.63	17.98	21.00
w $\mathcal{L}_{\text{SDFlow}}$	21.82	0.47	20.58	20.72	20.21	18.20	19.55	15.40	20.16	20.16	16.98	18.60	23.15
Glyph	a	b	c	d	e	f	g	h	i	j	k	l	m
Corners	8.83	8.28	5.47	8.04	8.04	11.55	8.67	10.17	6.43	6.42	13.12	5.46	13.08
w/o $\mathcal{L}_{\text{SDFlow}}$	22.66	21.03	19.55	21.35	22.04	19.47	21.94	18.40	14.80	18.91	18.77	14.70	21.14
w $\mathcal{L}_{\text{SDFlow}}$	21.55	20.54	19.44	20.14	22.16	19.79	21.03	19.23	14.41	17.58	18.81	14.87	20.56
Glyph	n	o	p	q	r	s	t	u	v	w	x	y	z
Corners	9.27	2.73	8.70	8.47	8.91	6.76	10.60	8.15	8.11	13.74	13.93	9.78	9.88
w/o $\mathcal{L}_{\text{SDFlow}}$	20.08	0.32	19.37	20.73	20.96	17.72	18.88	19.05	18.51	21.01	17.07	19.88	21.53
w $\mathcal{L}_{\text{SDFlow}}$	21.33	0.24	19.56	20.25	21.11	17.69	18.40	20.27	21.14	21.14	18.34	20.00	24.13

Table 2: Average number of corners for each alphabet in the dataset, Divergence w and w/o SDFlow loss.

Figure J

Figure K

Figure L

Figure M