CS230

font-gen: Deep Models for Inferring Alternate Language Sets from Fonts

Garrick Fernandez
Department of Computer Science
Stanford University
garrick@cs.stanford.edu

Abstract

Fonts display a large amount of stylistic variance; as a result, the task of creating new ones is a labor-intensive process usually reserved for skilled artists and designers. In this paper, we examine methods of generating characters for a language set from fonts which lack them—methods which, with refinement, could serve as useful tools for designers looking to "internationalize" existing and in-progress fonts quickly and easily. We show that a set of four "basis" characters encodes adequate stylistic information for the following three tasks: telling whether an novel character belongs to the font or not; generating a new Latin character (A-Z) in the style of the original font; and generating a new foreign language glyph (e.g., Japanese *hiragana*) in the style of the original font. We explore neural network architectures and techniques (e.g. multitask learning, transfer learning) for performing these tasks and compare their results.

1 Introduction

Typefaces (colloquially, and for the rest of this paper, *fonts*) are the designs of letters and other glyphs. In today's visual culture, they are ubiquitous and pervasive, and the choice of font is one that can greatly affect our understanding, emotion, and trust in the things we see.

It's no wonder, then, that there are thousands of fonts out there (one popular site, MyFonts [1], hosts over 130,000 of them!) with yet more being designed. Designing a stylistically consistent font, however, is a challenging task, and it only becomes more daunting with characters originating from a different language set (e.g. Japanese, which encompasses the *hiragana* and *katakana* syllabary as well as *kanji*, a set of logograms adapted from Chinese). Few fonts implement these character sets to be stylistically consistent with Latin because it is a daunting task to design and draw consistent glyphs for thousands of characters (one site, Free Japanese Fonts [2], hosts only 240 Japanese fonts).

In this paper, we examine methods of generating unique characters from a language set for fonts which lack them, which could have applications for designers looking to create these language sets from limited observation. The paper proceeds in three tasks:

1. In the first, we attempt to classify whether a random character belongs or does not belong to a set of 4 basis characters. The inputs are a set of four basis characters and a fifth, unknown character. All characters are represented as 64×64 grayscale images. The output is whether the fifth character belongs to the same font as the other four characters.
2. In the second task, we use the same 4 basis characters to infer just the Latin character set (here, the 26 uppercase Latin characters).

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

3. In the third task, we transition from generating characters from the Latin character set to generating those from another language set (here, a subset of 46 Japanese hiragana). This greatly reduces the amount of data we can train on.

2 Related work

This paper draws inspiration from Shumeet Baluja's work [3], from which we have adapted our first two tasks and the inspiration for some of the neural network architectures we present later in the paper. One such architecture uses separate towers to encode each character separately, then concatenates the resulting representations. The idea is that the towers would be better at capturing individualized character transformations, compared to a dense layer connected to all input characters. Baluja also found, though a review of multiple architectures, that (1) deeper networks did not necessarily perform better; (2) ReLU was more effective than sigmoid activation; and (3) networks using convolutions benefited from parameter sharing, but translation invariance wasn't crucial because the characters were mainly centered, and the results of networks using convolution were on par with dense layers.

Baluja did not specify numbers of layers and neurons, so we took some liberties in redesigning the networks. In addition, the paper does not discuss the task of generating characters that do not exist.

Azadi et al. do discuss the generation of novel characters, presenting a network that not only generates the shape of a glyph, but also its ornamentation (think of the way fonts are colored and decorated in movie posters) with a pair of conditional generative adversarial networks (cGANs) [4]. This can be useful for designers, who may only initially design a subset of the glyphs needed for a project, but later wish to adapt them to a new one. The idea of few-shot style transfer draws similarities with Baluja's basis characters (which can be interpreted as 4 few-shot observations of a font, with the rest of the characters missing), but the GAN architecture is different. It's possible the generator, when trained with the discriminator, could generalize better to novel glyph input.

Erik Bernhardsson wrote a blog post that served as the initial starting point for experimentation and data collection [5]. He outlined the process we adapted for extracting data and noted some insights in the training process that benefited him (larger batch sizes, Leaky ReLU activations, high learning rate, mean absolute loss for less "gray" images). However, he only iterated on one potential network design.

Hideaki Hayashi et al. designed a GAN that attempts to better the style consistency, legibility, and diversity of generated fonts [6]. The paper itself draws inspiration from Alec Radford et al.'s paper on Deep Convolutional GANs (DCGANs), which provides a framework of suggestions on making GANs with convolutional layers more stable and easier to train [7]. They also experiment with WGANs, a variation on GANs that uses the Wasserstein or earth mover's distance, which can be intuitively understood as the "cost" of moving a distribution (represented as a pile of dirt) to another. These approaches are interesting for their similarity to Azadi et. al.'s work, as well as the use of convolution, which Baluja argues doesn't provide significant benefit in the stacked perceptron/tower architectures.

Yet another approach by Raphael Gontijo Lopes et al. uses an encoder-decoder framework, but the latent representation in the encoder-decoder pair is fed to another decoder, which produces a scaled vector graphic (SVG) representation of the character [8]. This is interesting for the problem we wish to solve, because the scale invariance of the generated output would make less work for the designer, who can quickly adapt a vector into the final font design. The generation of an encoding is analogous to the towers Baluja describes, which also encode latent representations of input characters. The work is primarily concerned with learning SVG representations from raster (pixelated) counterparts however, and does not discuss, like Azadi et al., the generation of novel glyphs.

3 Dataset and Features

We adapted existing code by Erik Bernhardsson [9] to crawl a directory of font files and produce 64×64 pixel resolution, 8-bit grayscale images [10] of individual characters. The characters in each font are sized and centered in a way that preserves the relative vertical alignments of individual characters (maintaining how, for example, the descender of a 'g' drops below the baseline, while the

ascender in a 't' rises above the x line). The fonts themselves were taken from the Google Fonts repo [11], which offers 2908 free and open-source fonts in various styles and weights.

Additional scripts grouped individual characters as needed for each learning task and dumped training examples into HDF5 files [12]. For the discrimination task, 22 'in-font' ($y = 1$) and 22 'not-in-font' ($y = 0$) examples were generated for each font by sampling random characters and pairing them with the basis letters. For the two generative tasks, one example was produced for each font, since the basis letters are used to infer the entire set of 26 uppercase Latin characters (in the first generative task) or 46 Japanese hiragana (in the second). We used a roughly 75/15/15 train/val/test split (examples were assigned groups on-the-fly with a seeded random number generator), which resulted in the following dataset statistics:

			Data format		Number of examples		
task	source	size	x shape	y shape	train	val	test
1	fonts-50	45.3MB	(5, 64, 64)	(1,)	1543	325	332
2	fonts-all	357.8MB	(4, 64, 64)	(26, 64, 64)	2030	432	446
3	fonts-all-jpn	6.3MB	(4, 64, 64)	(46, 64, 64)	22	6	3

As seen above, there are only eight font families in Google Fonts that support Japanese, which makes the data available for that task scant.

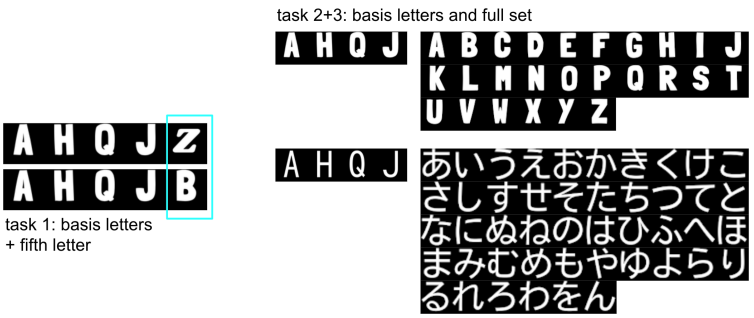

Figure 1: Examples of data.

4 Methods

We focused on variants of layered perceptrons for this paper due to their successful use in Baluja's paper, the ease of doing more iterations of architecture search, and the potential for the discriminative task's network to function as an encoder, producing a representation that can be fed to a series of layers in the generative task.

For the first task, we tried two architectures based on the tower design suggested by Baluja, one with densely connected layers in the towers, as well as one with convolutional layers. We also developed one "shared" architecture with convolution, to see if individual filters extracting similar features from all input characters could be useful to the network. After being fed through these initial layers, the activations are fed through a series of dense layers and a final layer containing one neuron with sigmoidal activation (to predict labels). The network was trained on binary cross entropy loss.

For the second task, we picked the tower architecture based on its high accuracy and low loss (see EXPERIMENTS) and tried several architectures that would produce all 26 uppercase Latin characters from four basis characters. In all variants, we used shared layers working to produce all characters, with the idea that common features between individual characters would help the network learn to produce all of them better. In contrast with other papers, which used mean absolute error, we used mean squared error in training, because we felt that higher penalties for outliers were appropriate for the pixel differences between the generated output and the real output. We also experimented with the presence of convolutional layers applied to the output.

For the third task, we picked the best of these architectures and tried adapting it to generate Japanese hiragana. We altered either training the network from scratch, or providing it the weights from the previous task, to see if transfer learning would help the network learn on the smaller set of data.

5 Experiments/Results/Discussion

All networks were trained in Keras [13][14][15] using an Adam optimizer with default parameters set. We iterated more on different architectures than on these hyperparameters. We used a minibatch size of 512, following Bernhardsson's suggestion that larger minibatch size gave better results.

For the first task, we recorded loss and accuracy, and we also looked at the false positive and false negatives for our chosen network. The tower network with dense layers outperformed the other two networks. There were more false negatives than positives, which suggested that the network leaned towards guessing no a lot of the time.

Experiment			BCE Loss			Accuracy		
#	description	parameters	train	val	test	train	val	test
1	tower, dense	330K	0.0974	**0.6513**	0.2770	96.24%	**88.31%**	89.76%
2	tower, conv	6.2M	0.4591	0.6595	-	76.22%	77.54%	-
3	shared, conv	285K	1.3451	1.4176	-	76.09%	62.15%	-

$n = 332$		Predicted	
		$y = 1$	$y = 0$
Actual	$y = 1$	159	23
	$y = 0$	11	139

for Experiment 1		
Precision	Recall	F1 Score
0.935	0.874	0.903

Figure 2: First task losses, accuracy; confusion matrix and statistics for experiment 1.

For the second task, we recorded loss as well as the generated images at each epoch. The best network had more neurons, as well as one convolutional layer in the output that had the effect of "smoothing" the output. Additional layers did not seem to have as beneficial an effect.

Experiment			MSE Loss		
#	description	parameters	train	val	test
1	1 extra conv layer	21.7M	1560.2068	2261.4195	-
2	no conv layer	21.7M	1902.4307	2645.9201	-
3	x2 neurons	43.4M	1380.6402	**2182.6619**	1696.2317
4	x2 decoder neurons, add layers	43.3M	1683.7724	2391.8688	-
5	x2 encoder neurons, add layers	22.1M	1598.5508	2324.1624	-
6	x2 neurons, add layers	43.6M	1432.9696	2223.8934	-

Figure 3: Loss for second task

Figure 4: Comparing generated output with and without convolution.

The third task proceeded like the second; we also fed it a set of fonts with no Japanese set to see what would happen! Some fonts, like Times New Roman and Futura, generalized well, while others, like Cooper Black and Comic Sans, were too novel for the network to generalize to.

Experiment			MSE Loss		
#	description	parameters	train	val	test
1	no transfer learning	76.3M	1049.1206	1489.6694	-
2	transfer learning	76.3M	750.9727	**1427.2902**	1459.3241

Figure 5: Loss for third task.

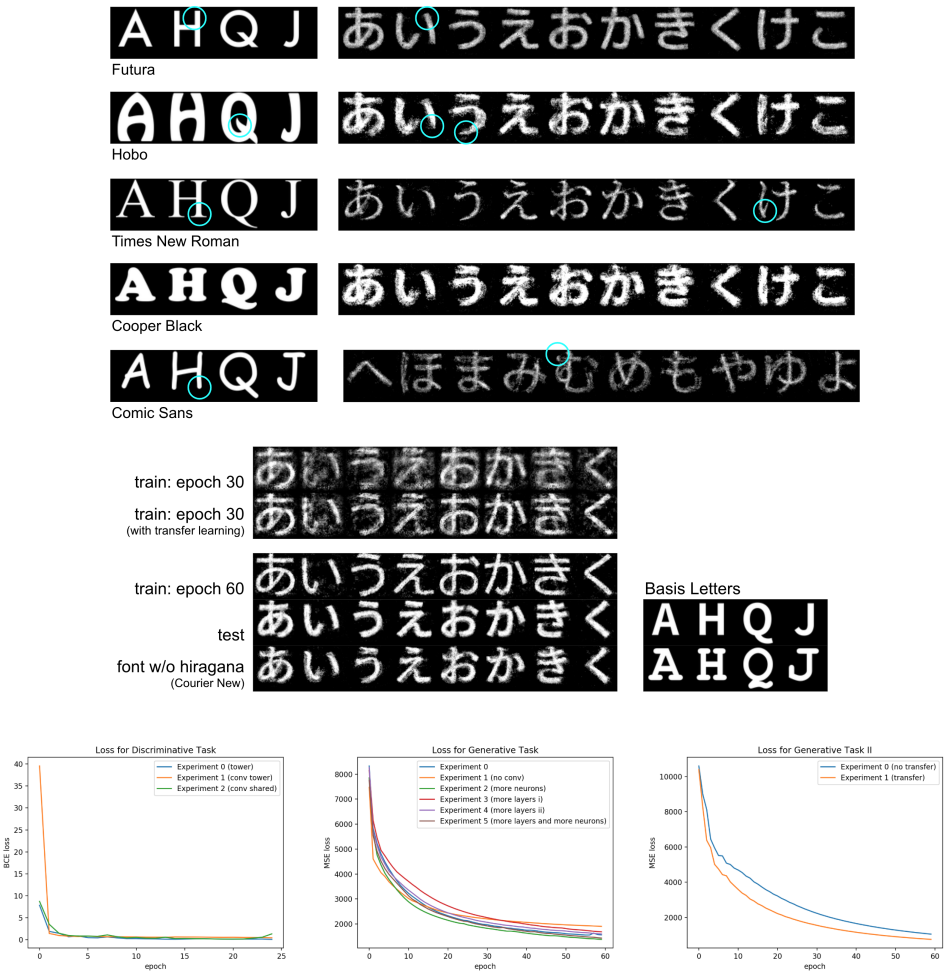

Figure 6: Top to bottom: predictions for novel fonts, highlighting similar features, training, loss plots across all tasks.

6 Conclusion/Future Work

In conclusion, multitask learning can be a good approach to generating characters due to shared features. The tower network with a more neurons works well to encode four basis characters, and convolutional layers get rid of a grainy effect in generated letters. Transfer learning works well in the third task, with information about Latin characters carrying over well to hiragana.

Some issues with this approach are that similar fonts can have different/distinguishing marks that would be hard to generalize to, some fonts are not represented in Japanese (see the monospaced-serif font Courier New or the more hand-drawn Comic Sans).

With more time, team members, or computational resources, we would have liked to explore extensions of the network to generate more characters, or variational autoencoders (VAE) and GANs.

References

[1] Monotype, MyFonts. URL: https://www.myfonts.com/

[2] Free Japanese Fonts. URL: https://www.freejapanesefont.com/

[3] S. Baluja, Learning typographic style: from discrimination to synthesis, arXiv preprint arXiv:1603.04000

[4] S. Azadi, M. Fisher, V. Kim, Z. Wang, E. Shechtman, T. Darrell, Multi-content GAN for few-shot font style transfer, arXiv preprint arXiv:1712.00516

[5] E. Bernhardsson, Analyzing 50k fonts using deep neural networks. URL https://erikbern.com/2016/01/21/analyzing-50k-fontsusing-deep-neural-networks.html

[6] H. Hayashia, K. Abea, S. Uchidaa, GlyphGAN: style-consistent font generation based on generative adversarial networks, arXiv preprint arXiv:1905.12502

[7] A. Radford, L. Metz, S. Chintala, Unsupervised representation learning with deep convolutional generative adversarial networks, arXiv preprint arXiv:1511.06434.

[8] R. Gontijo Lopes, D. Ha, D. Eck, J. Shlens, A learned representation for scalable vector graphics, arXiv preprint arXiv:1904.02632

[9] E. Bernhardsson, deep-fonts. URL https://github.com/erikbern/deep-fonts

[10] Pillow. URL https://pillow.readthedocs.io/en/stable/

[11] Google, fonts. URL https://github.com/google/fonts

[12] HDF5 for python. URL https://www.h5py.org/

[13] Tensorflow. URL https://www.tensorflow.org/

[14] Keras, the python deep learning library. URL https://keras.io/

[15] G. Fernandez, font-gen. URL: https://github.com/garrickf/font-gen